Chair of
Software Engineering

Modelling and Verifying an Object-Oriented Concurrency Model in GROOVE

Master's Thesis

Claudio Corrodi
ETH Zurich
clcorrod@ethz.ch

October 10, 2014 – April 10, 2015

Supervised by:
Christopher M. Poskitt & Alexander Heußner
Prof. Bertrand Meyer

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

inf | Informatik
Computer Science

In memory of Renato.

Abstract

SCOOP is a programming model and language that allows concurrent programming at a high level of abstraction. Several approaches to verifying SCOOP programs have been proposed in the past, but none of them operate directly on the source code without modifications or annotations.

We propose a fully automatic approach to verifying (a subset of) SCOOP programs by translation to graph-based models. First, we present a graph transformation based semantics for SCOOP. We present an implementation of the model in the state-of-the-art model checker GROOVE, which can be used to simulate programs and verify concurrency and consistency properties, such as the impossibility of deadlocks occurring or the absence of postcondition violations. Second, we present a translation tool that operates on SCOOP program code and generates input for the model. We evaluate our approach by inspecting a number of programs in the form of case studies.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

___

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Modelling and Verifying an Object-Oriented Concurrency Model in GROOVE

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Corrodi	Claudio

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zürich, April 10, 2015	*C. Corrodi*

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.

Contents

Acknowledgments

I would like to thank Chris Poskitt, Alexander Heußner, and Bertrand Meyer for supervising this thesis, giving me the opportunity to work on this project, and providing helpful input in meetings and conference calls. In particular, I would like to thank Chris Poskitt for providing advice and support in countless discussions, meetings, and email conversations during the past six months. I would also like to thank Benjamin Morandi for his input and suggestions.

Claudio Corrodi, April 2015

Author's Declaration

I declare that the work and results presented in this thesis are my own, except where otherwise stated. Parts of this thesis have been published in [8], where I am a contributing author. Details regarding the use of the results of this publication can be found in Section 1.4.

Claudio Corrodi, April 2015

Chapter 1

Introduction

In this chapter, we start with the motivation of this thesis and describe our contributions, before we present the research hypothesis and list the goals that we want to meet in order to consider this thesis a success. An overview of the thesis structure follows, before we close this chapter with a description of previously published work present in this thesis.

1.1   Motivation

With the shift to multiprocessor and multicore systems, concurrent and parallel programming becomes an important part of object-oriented software development. While object-oriented models and languages allow programming at a high level of abstraction when writing sequential programs, they often rely on low-level constructs for concurrency, such as locks, semaphores, and threads. These constructs are very error prone and difficult to use correctly. *Simple Concurrent Object-Oriented Programming* (scoop) is a concurrency model and language that extends Eiffel with concurrency mechanisms. The model hides low-level constructs in its implementation, and instead provides the user with simple to use constructs that allows concurrency to be expressed at a high level of abstraction. In particular, lock management and thread creation is no longer expressed explicitly.

It is still possible to introduce concurrency bugs with the high-level constructs from scoop, most prominently deadlock. Naturally, these bugs are difficult to detect, as they may not occur in every program execution. With program verification approaches, it is possible to prove the correctness of implementations and make sure that no concurrency related bugs exist for the modelled semantics and input program.

Currently, there exist several formalisations of scoop [24, 14, 1, 16]. They do not focus on verification but rather resolving language ambiguities. They can not be used for model checking, due to the state-space explosion problem inherent to concurrency models. In addition, existing approaches for the verification of scoop programs [3, 23] focus on deadlock prevention and work on either annotated source code or with manually translated model input.

In this thesis, we propose an alternative approach to verify scoop programs. First, we present a graph-based model that focuses on the concurrency mecha-

nisms in SCOOP, leaving out advanced object-oriented features. This leads to a compact model with a strong formal foundation. Second, we add object-oriented features from SCOOP to the model, obtaining an expressive formalisation that allows representing SCOOP programs more directly as model input. The models are implemented using the state-of-the-art model checker GROOVE. We then present a translation tool that works directly on SCOOP source code. With this tool, we are able to translate a subset of SCOOP programs and generate input for the model checker. By putting these parts together, we provide a fully automatic tool that allows verification by model checking. We focus on verification of properties like deadlock or pre- and postcondition violations. By focusing on the core of SCOOP and abstracting away from internals of the formalisations, we are able to reduce the state-space sizes. We discuss why our abstractions and optimisations do not change the expressiveness of the modelled SCOOP subset.

1.2 Research Hypothesis and Contributions

The research hypothesis is as follows.

> A subset of valid SCOOP programs can be modelled using a graph transformation system. These programs can, without modification of the source code, be automatically translated to input graphs for the transformation system. Using verification by model checking, it is possible to verify a number of properties such as absence of deadlock or absence of precondition violations for a given input program.

To satisfy the hypothesis, we specify the following goals for this project:

- Provide a formalisation of a subset of SCOOP as a graph-based model using the GROOVE toolkit.

- Create a translation tool that operates on SCOOP source code and generates input graphs for the model.

- Make informal soundness arguments for the translation and model.

- Provide a simple tool that allows verification of certain properties like deadlock freedom or absence of precondition failures with a single step by specifying SCOOP source code and model parameters.

- Evaluate the created translation tool and graph model by inspecting a number of SCOOP programs that use its concurrency features, as well as a thorough discussion of the characteristics and performance of the toolchain.

1.3 Thesis Overview

Chapter 2 gives an overview of the SCOOP model and its primary implementation.

Chapter 3 briefly describes the theoretical background of this thesis and gives a detailed description of GROOVE, the main tool used to implement the graph models.

Chapter 4 presents a formal model that focuses on the fundamental concurrency mechanisms of SCOOP and gives an in-depth description of its implementation in GROOVE.

Chapter 5 builds on the previous one by extending the model with object-oriented features and discussing the extended model in detail.

Chapter 6 discusses the design and implementation of a translation tool that translates scoop programs to input graphs for the model described in Chapter 5. A discussion of future work with respect to Chapters 5 and 6 closes this chapter.

Chapter 7 evaluates our implementations, with a focus on the model described in Chapter 5. We discuss a number of examples in depth, where we present source code, generated model input, and obtained results. We look at the performance of our model in conjunction with GROOVE from several angles.

Chapter 8 concludes this thesis by summarising our contributions and revisiting the research hypothesis.

The tools and programs that were written during the course of this project can be found online at [21].

1.4 Published Work

This thesis contains results published in the following paper, where the author of this thesis is a contributing author.

[8]   Alexander Heußner, Christopher M. Poskitt, Claudio Corrodi, and Benjamin Morandi. "Towards Practical Graph-Based Verification for an Object-Oriented Concurrency Model". In: *Proc. Graphs as Models (GaM 2015)*. Electronic Proceedings in Theoretical Computer Science. To appear. 2015.

This thesis contains results from the paper as follows.

- Chapter 4: The *Concurrent Processor Model* (CPM) (which was developed with Poskitt and Heußner) is presented in the paper. The detailed description of its GROOVE implementation is my own work.

- Chapter 5: A brief overview of CPM *with Object Orientation* (CPM+OO) has been given in the paper in Section 5, which I contributed to. In this thesis, we present the model in more detail. The augmented model in this chapter is my own work.

- Chapter 7: Programs presented in the paper are reused in this thesis. The CPM+OO model has undergone several changes since writing the paper, and the results presented in this thesis are based upon more recent revisions of the CPM+OO model.

Chapter 2

SCOOP: An Object-Oriented Concurrency Model

Simple Concurrent Object-Oriented Programming (SCOOP) [15] is a programming model that provides concurrent, asynchronous, object-oriented programming. Its main implementation is an extension to the Eiffel programming language and is distributed with the EiffelStudio[1] software. Several formalisation of the model exist, with the most recent by Morandi [13], which we consider in this work.

Basic knowledge of the Eiffel programming language, along with concepts like Design By Contract, is assumed. A general introduction to the language and its concepts can be found in [12].

In this chapter, we give an overview of the SCOOP model and introduce a running example for the remainder of this thesis.

2.1 The SCOOP Model

The goal of SCOOP is to enable concurrent programming at a high level of abstraction, without relying on low-level constructs like locks, semaphores, and threads. To achieve this, SCOOP adds a new keyword `separate` to the Eiffel language, which allows expressing concurrency relations between objects as follows.

SCOOP introduces the notion of a *processor*, which is an abstract thread of execution that is able to execute instructions sequentially. A processor is the *handler* of a number of objects, and object references can point to objects that are handled by the same processor (*non-separate* references) or objects that are potentially handled by different processors (*separate* references). The set of objects handled by a given processor is called a *region*.

In the source code, one can annotate types (in particular in feature declarations and formal arguments) as `separate`, expressing that the reference points to an object potentially handled by a different processor.

---

[1] `https://www.eiffel.com/eiffelstudio/`, accessed April 10, 2015

With the concept of processors, the semantics of feature calls are different. If a client executes the call `a.f(b1, b2, …)`, with target `a` and arguments `b1, b2, …`, then the following cases can be distinguished:

- If the target `a` is handled by the current processor, then the call is applied immediately.

- If the target `a` is handled by a different processor, then the client logs the call with the supplier. The call is then enqueued in the *request queue* of the handler of the supplier and processed at some point in the future.

In the second case, depending on whether the call is a *command* (a call that does not return a value) or a *query*, the client waits for the supplier to execute the request. In the first case, the client can continue execution without waiting. In the second case, the client needs the result (e.g. as a value in an assignment, or as a value for parameter passing), and therefore waits until the supplier returns the value, making the call sequential.

In order to avoid data races, SCOOP only allows calls on separate targets which are formal arguments of the enclosing routine. When executing a routine, the SCOOP runtime waits for exclusive access to the request queues of the handlers of the separate arguments. Once the request queues are locked, the routine starts executing and, since no other processor has access to the locked request queues, the requests logged by the routine are guaranteed to be executed in order and without interleaving requests from other processors. Shared memory, another source of data races, does not exist in SCOOP, since object data can only be modified using procedures and not directly accessed from outside (e.g. a statement like `foo.id := 0` is forbidden if `id` is an attribute).

Contracts, i.e. class invariants and routine pre- and postconditions are an integral part of Eiffel. Preconditions are Boolean assertions that must hold before the body is executed. If a precondition does not hold, a runtime error occurs. In SCOOP, the semantics of preconditions change. While statements involving non-separate objects behave like before, expressions involving separate objects can become *wait conditions*. If a wait condition does not hold, the processor simply waits until it holds instead of generating a runtime error. For example, a consumer might have a precondition in its `consume` routine that states that the inventory must have an item ready, as seen in Listing 2.1. Since the inventory is separate and its state can be modified through requests from other processors, the consumer simply waits until the inventory is not empty anymore. Locks are acquired before the preconditions and wait conditions are evaluated, but released if a wait condition does not hold yet, giving other processors the possibility to enqueue request on the handlers of the targets. Wait conditions are a powerful and expressive synchronization mechanism. The lack of explicit locking makes this kind of synchronization particularly easy to use.

```
1  class CONSUMER
2  feature
3    consume (a_inventory: separate INVENTORY)
4        -- Consume the item held in `a_inventory'.
5      require
6        full_inventory: a_inventory.has_item
7      do
8        consumed_item := a_inventory.item
9        a_inventory.remove
```

```
10      end
11
12   -- remaining class code omitted
13 end
```

Listing 2.1: `CONSUMER.consume` routine implementation.

2.2   A Running Example

Throughout this report, we will use a running example to demonstrate the contributions of this project, in particular translation to and verification with our formal models in GROOVE.

The *Dining Philosophers Problem* is a well known problem that involves several entities interacting and is well suited for demonstrating concurrency models. In this problem, a number of philosophers sit at a round table, with a fork in between each pair of adjacent philosophers. The philosophers each perform two activities in a loop: thinking and eating. In order to eat, a philosopher needs to pick up both the left and the right fork before eating, and put them down afterwards. The goal is to devise an algorithm that abides these rules and does not get stuck in a deadlock.

Listing 2.2 shows the `PHILOSOPHER` class of a SCOOP implementation (which we adapted from an implementation in the EVE [22] source code repository) of the problem. During his time at the table (feature `live`), a philosopher eats `times_to_eat` times. Notice how there is no code handling picking up and putting down the forks. Instead, this is done implicitly: the `eat` routine takes two objects of type **separate** `FORK` as arguments. Once a philosopher is inside the (empty) `eat` body, he has exclusive access to the processors handling the left and right forks, simulating picking up both forks and thus not allowing other philosophers getting access to the forks. Since both forks are arguments in the same routine, their respective processors are locked atomically, which guarantees that no deadlock can occur.

While SCOOP allows concurrent programming at a high level of abstraction, it can still be difficult to spot problems related to concurrency. For example, an unexperienced SCOOP programmer may have implemented the `eat` method as shown in Listing 2.3. In this implementation, a philosopher first picks up the left fork, and then the right one. An execution may take place where each philosopher picks up its left fork and waits for the right one to become available, which never happens; the program is stuck and a deadlock has occurred.

```
1 class
2    PHILOSOPHER
3
4 create
5    make
6
7 feature
8
9    make (philosopher: INTEGER; left, right: separate FORK;
          round_count: INTEGER)
10        -- Initialise with ID of `philosopher', forks `left' and `
              right', and for `round_count' times to eat.
11       do
12        -- (initialization left out)
```

```
13        end
14
15     id: INTEGER
16          -- Philosopher's id.
17
18     times_to_eat: INTEGER
19          -- How many times does it remain for the philosopher to eat?
20
21     eat (left, right: separate FORK)
22          -- Eat, having acquired `left' and `right' forks.
23        do
24          -- Eating takes place.
25        end
26
27     live
28        do
29          from
30          until
31            times_to_eat < 1
32          loop
33            -- Philosopher `Current.id' waiting for forks.
34            eat (left_fork, right_fork)
35            -- Philosopher `Current.id' has eaten.
36            times_to_eat := times_to_eat - 1
37          end
38        end
39
40     left_fork: separate FORK
41          -- Left fork used for eating.
42
43     right_fork: separate FORK
44          -- Right fork used for eating.
45
46  invariant
47     valid_id: id >= 1
48
49  end
```

Listing 2.2: Implementation of a philosopher in SCOOP.

```
1  bad_eat
2      -- Eat, by first picking up `left_fork' (and picking up `
           right_fork'
3      -- in the subsequent `pickup_right' call).
4    do
5      pickup_left (left_fork)
6    end
7
8  pickup_left (left: separate FORK)
9      -- After having picked up `left', proceed to pick up `
           right_fork'.
10   do
11     pickup_right (right_fork)
12   end
13
14 pickup_right (right: separate FORK)
15     -- Both forks have been acquired at this point.
16   do
17     -- eating takes place
18   end
```

Listing 2.3: Implementation of the `eat` feature that can result in a deadlock.

2.3 Related Work

A first description of SCOOP appeared in 1993 [11] and an updated description was published in 1997 [10]. A prototype of SCOOP has been implemented between 2005 and 2008 at ETH Zürich, and an implementation maintained by Eiffel Software[2] is currently distributed with the EiffelStudio IDE.

Since its introduction, several formalisations of SCOOP have been proposed [2, 14, 16, 24, 15]. We consider the work done by Morandi et al. [14, 13] in this thesis.

---

[2]`https://www.eiffel.com/`, accessed April 10, 2015

Chapter 3

Graph Transformation Systems & GROOVE

In the course of this project, we have been working with the *GRaphs for Object-Oriented VErification* (GROOVE) [19] toolkit, which is a set of tools based on a strong formal foundation in *Graph Transformation Systems* (GTS) that can be used for modelling, simulation, and verification. In this section, we give a short informal introduction to the GTS theory GROOVE is based on and then discuss the GROOVE toolkit in detail. We showcase its features by providing a GTS for our running example, the dining philosophers problem.

A graph transformation is, informally speaking, the process of altering an input graph to get an output graph by using rules that describe the manipulation. There are a number of different approaches to graph transformation, which provide a wide range of semantics of rule applications. From an operational standpoint, the approaches differ in how rules are defined and in the situations in which they can be matched and applied. One such approach is the algebraic approach, discussed in [5], which is used by the GROOVE toolkit.

3.1 The Algebraic Approach

In the algebraic approach to graph transformation systems, pushout constructions (from category theory) are at the core and are used to allow gluing graphs together. The two main approaches, *Double-Pushout* (DPO) and *Single-Pushout* (SPO), allow for a compact and abstract representation of graph transformations. What follows is an informal overview of these approaches.

The DPO Approach   In the DPO approach, a graph transformation is described as a *rule* consisting of three graphs, L, K, and R. The graph K describes the interface of the rule, i.e. the parts of the graph to be matched and preserved. The left-hand side $L \backslash K$ of the rule describes the part that is to be deleted, and the right-hand side $R \backslash K$ the part that is to be created. An application of the rule described by L, K, and R on the graph G, shown by example in Figure 3.1, is performed by applying the following steps.

1. Find a *morphism* from L to G, that is, find a structure-preserving mapping
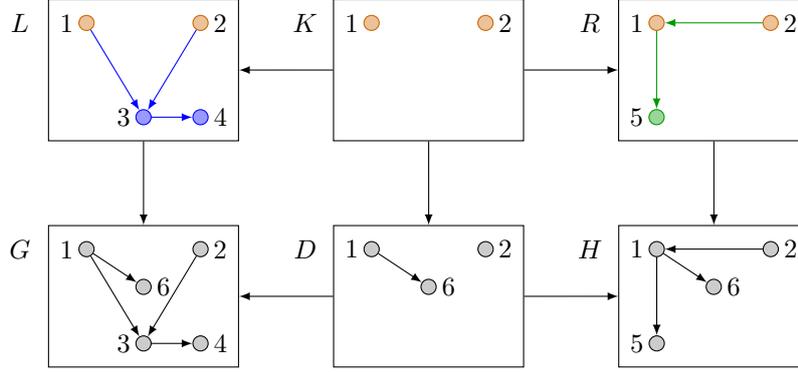
Figure 3.1: Rule application in the DPO approach.

from nodes and edges in L to nodes and edges in G. In Figure 3.1, this mapping is expressed by node identifiers, where nodes in L are mapped to nodes in G with the same identifier.

2. Construct a graph D from G by removing the matched edges and nodes in $L\backslash K$ from G. The combination of L and D at the interface nodes and edges from K (in our example nodes 1 and 2) is called *glueing* and results in G.

3. With a similar combination of D and R using the interface K, the output graph H is obtained.

The SPO Approach   In the SPO approach, specifying K is omitted. Instead, a rule consists only of the left-hand side L and the right-hand side R. An example application in the SPO approach is shown in Figure 3.2. The following steps are necessary to apply a rule in the SPO approach.

1. Obtain the common interface $K = L \cap R$.

2. Find a morphism from L to G, as in the DPO approach.

3. Delete $L\backslash K$ from G, and join $R\backslash K$, using the common interface as glueing nodes and edges. If there are *dangling edges* (i.e. edges that have a source or a target, but not both) remaining, delete them as well.

They key difference between SPO and DPO is that in the SPO approach, dangling edges are allowed in the final graph, which is not possible in the DPO approach. Figure 3.2 shows a situation where a dangling edge (the edge between nodes 3 and 4 in G) remains after deleting $L\backslash K$, which is then deleted as well.

GROOVE allows configuring whether applications which delete dangling edges are allowed. If so, dangling edges simply get deleted after the application to make sure the resulting construct is a valid graph. Otherwise, when only cases without dangling edges are allowed, SPO and DPO are equivalent form an operational point of view. In our models, we never allow applications with dangling edges, which requires us to specify all edges incident to a deleted node on the left-hand side of a rule, ensuring that no edges get deleted "by accident".
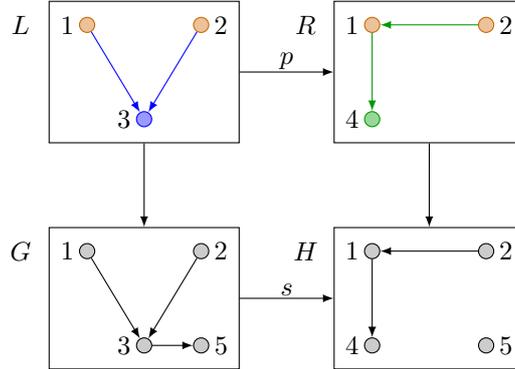
Figure 3.2: Rule application in the SPO approach. Orange nodes denote the common interface K, blue edges and nodes the parts of L that are to be deleted, and green edges and nodes of R the ones that are to be added. Note that the edge between nodes 3 and 5 in G is a dangling edge after the deletion of $L\backslash K$ in G, and is therefore deleted as well.

3.2   GROOVE

The GROOVE toolkit is written in Java and consists of a number of components.

The **Simulator** is a GUI tool that provides features to create and edit *Graph Production Systems* (GPS). It is particularly useful for designing a system as it provides immediate feedback on how the system behaves. One can apply rules to start graphs and explore a *Labelled Transition System* (LTS) either by manually choosing rules to apply one after another, or by automatically exploring the state-space for a certain amount of applications.

With a finished GPS, using the Simulator to model check various start graphs can become cumbersome and automating the task becomes difficult. For this scenario, the **Generator** was created, which is a command-line tool that explores the state-space of a given GPS. Like the Simulator, the Generator can use different strategies for exploration, such as breadth-first-search or depth-first-search. The Generator also allows specifying *Linear Temporal Logic* (LTL) and *Computational Tree Logic* (CTL) formulae (a thorough discussion of LTL and CTL can be found in [9]) and searching for counterexamples. The Generator provides various metrics such as the size of the LTS, feedback about LTL and CTL formulae, and profiling information.

Other components that can be used as standalone applications are included in the above two tools. The **Model Checker** can be used to verify LTL and CTL properties for labelled transition systems created by the Generator, but is included in the Generator as well. The **Viewer** is a simple GUI tool that can render graphs from a GPS and is used as part of the Simulator.

3.2.1   Graph Production Systems

GROOVE stores its Graph Production System in `.gps` folders. Such a folder consists of the following components (stored as individual files).

- Production rules are stored as `.gpr` files and encode graph transformations

in the sense of the SPO approach. They are rendered as a single graph using colour codes to distinguish left-hand side, interface, and right-hand side, as well as other properties of the rule.

- Type graphs are stored as `.gpy` files. If active, GROOVE only allows using rules and start graphs that conform to them. Multiple type graphs can be active at a time.

- Start graphs are stored as `.gst` files and represent stating points for the exploration.

- The `system.properties` file contains a number of configuration entries, most notably whether dangling edges should be allowed, the name of the active start graph, the active type graphs, and the exploration strategy to be used.

  An important system property is whether rules can be matched injectively or not. If so, distinct nodes that are matched from a source must have a distinct node in the target graph. Otherwise, multiple nodes in the rule can be mapped to the same node in the target graph. The configuration of this property can be overridden for individual rules.

The individual files conform to the *Graph eXchange Language* (GXL) file format, which is an XML format that specifies graph information. It is used in GROOVE to store individual graphs and associated properties in the files mentioned above. Using an XML representation of GPSS makes pre- and postprocessing of GROOVE input and output respectively very accessible and easy to handle.

To illustrate how the various components of a GPS work together, we model the dining philosophers problem as a simple GPS in GROOVE. Note that the representation in this section is unrelated to SCOOP or the formal models we introduce in Chapters 4 and 5, and instead is a standalone model of the problem.

3.2.2 Type Graphs

Type graphs determine the form of other graphs in the system, in particular the form of rules and start graphs. While the feature is optional, it is rather useful when working on a system, as graphs that do not conform to the specified type graph are highlighted in the Simulator, which helps to detect typos and other errors.

Figure 3.3 shows the type graph for a GROOVE solution to the dining philosophers problem. It specifies that a philosopher can be *hungry* (using an optional node flag) and has a hunger integer value attached. The only edges in this system are edges from philosophers to forks. Not only has a philosopher edges to its left and right forks, but it can also have a lock on them, expressed by the lock edge, indicating that a philosopher has picked up the forks.

3.2.3 Graph Representation

The GROOVE Simulator augments graph representations from a simple directed graph with edge labels to a more compact, readable format. As mentioned earlier, rules are represented as one single graph with nodes and edges of different kinds
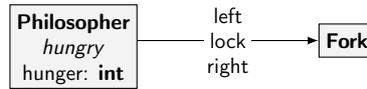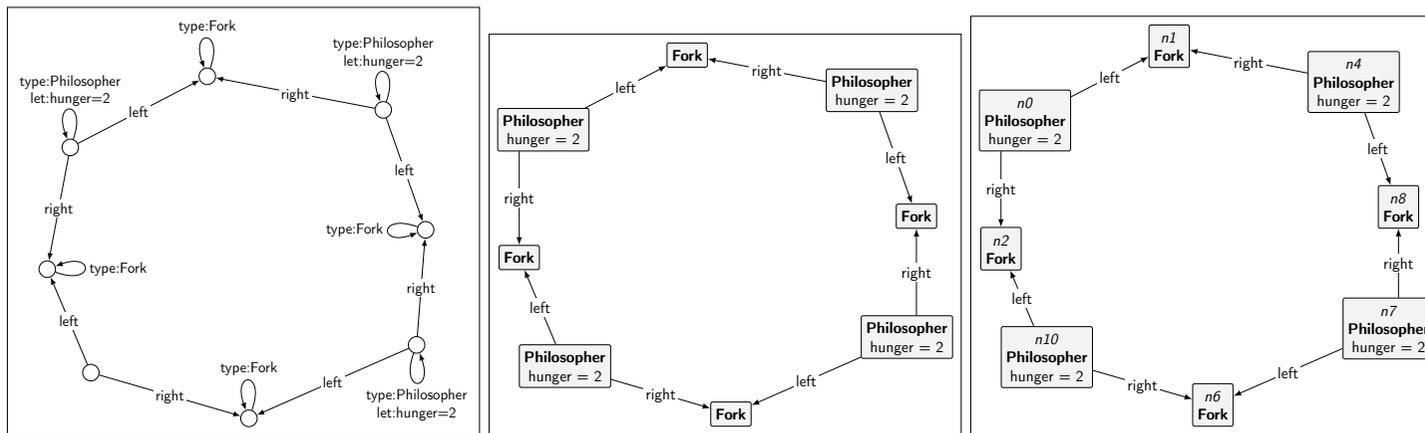
Figure 3.3: Type graph of the dining philosophers GTS.

of nodes and edges (in particular, readers, erasers, creators, embargoes, and conditional creators).

Figure 3.4 shows the start graph for a configuration of the dining philosophers problem with four philosophers. On the left-hand side, the start graph is shown as a directed graph with labelled edges. In the middle, the condensed form that GROOVE uses is shown, where self-edges are collapsed into the nodes, and on the right-hand side the graph is rendered in GROOVE with internal node identifiers (note that they are not a part of the model). We use the GROOVE representations of graphs throughout this report, as the resulting graphs are intuitive and contain the same information as before. We occasionally enable node identifiers in order to be able to refer to individual nodes easily.

In this model of the dining philosophers problem, each node that has a self-edge labelled `type:Philosopher`—we say "a node of type `Philosopher`" in this case—has two outgoing edges to nodes of type `Fork`, one of them labelled `left` and the other one labelled `right`. In addition, philosophers contain an integer value denoting the amount of times a philosopher wants to eat, encoded by the self-edges labelled `let:hunger=2`. The goal is to find rules that model the behaviour of the philosophers, namely grabbing the forks, eating, and putting them back down.

(a) Dining Philosophers start graph as a directed graph with edge labels.

(b) The same start graph, as rendered by GROOVE.

(c) Start graph rendered in GROOVE with node identifiers.

Figure 3.4: Comparison of a start graph as a directed graph with edge labels (left) and as rendered in GROOVE (middle and right). In GROOVE, self-edges are collapsed and displayed inside the node. In addition, certain values are rendered differently, e.g. type: prefixes are omitted but the following value is printed in bold.
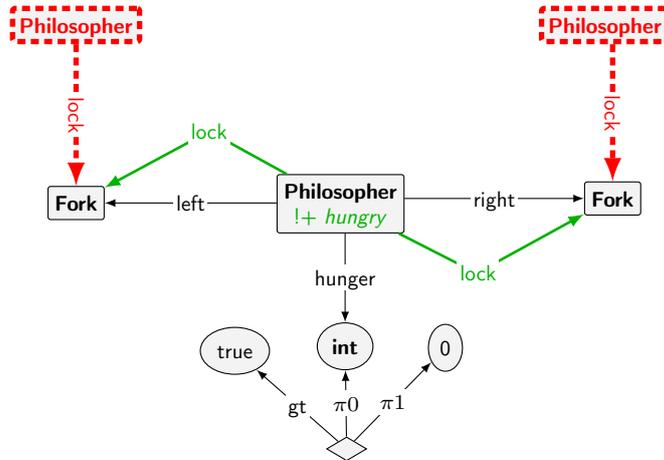
Figure 3.5: Picking up forks. A philosopher with positive hunger locks both forks if none of them is locked by another philosopher. The *hungry* flag is created to indicate that the philosopher has not eaten yet during this round.

Figures 3.5, 3.7, and 3.6 show the rules for picking up the forks, eating, and putting down the forks. These rules showcase various kinds of node and edge types, in particular the following.

Reader Edges and nodes that are displayed in black are the ones that are present in both sides of the rule. These are matched and preserved when applying the rule.

Creator Creator edges and nodes are only present in the right-hand side of the rule, which means that they are not required for the rule to match but will be created upon application.

Embargo These edges and nodes express a negative application condition. The rule only matches, if there is no match for these edges and nodes in the source graph. For example, in the pick up rule (Figure 3.5) , we express with embargo nodes and edges that a philosopher should only lock both forks if they are not already locked.

Eraser Finally, eraser edges and nodes (dashed blue) are elements that are only present on the left-hand side of the rule, which means that they are required for matching but will be deleted when the rule is applied.

Operations Arithmetic operations can be performed using product nodes (rhombus shaped nodes), which take a number of arguments (via π edges) and point to a result node (operation edge, such as `gt` for greater than). Figure 3.5 shows how the greater than operation can be used to enforce that the rule only matches if the hunger value is greater than zero.

More complex statements can be made with the help of *quantifiers*. Suppose the philosophers want to leave the table after having eaten the number of times that they wanted to. Since philosophers are polite, they do not leave until everyone at the table has finished. Figure 3.8 shows a rule that achieves this.
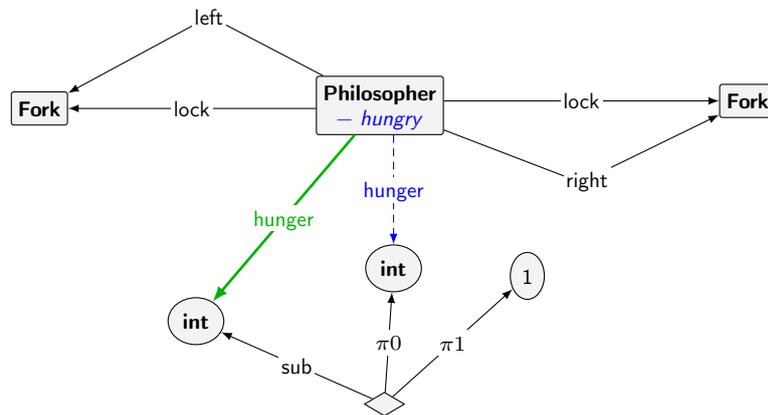
Figure 3.6: A philosopher eats if he is hungry. In the process, the *hungry* flag
(self-edge with label flag:hungry) is removed, and a product node is used to
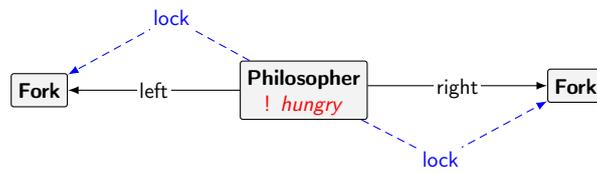decrease the hunger value by one.

Figure 3.7: A philosopher puts down his forks if he has them in his hands (lock
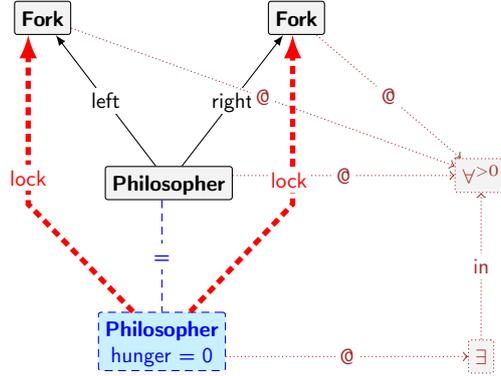edges) and is not hungry (embargo on the *hungry* flag).

Figure 3.8: The rule for leaving the table matches, if all philosophers (expressed by the forks and the philosopher reader node) are in a state where the hunger is zero and they do not have locks on forks, in which case the **Philosopher** gets deleted.

We first match all **Philosopher** nodes that have a left and a right fork attached with the $\forall^{>0}$ quantifier (which denotes that in order for the rule to apply, the subgraph "at" this quantifier (attached via @ edges) has to match at least once, as opposed to the $\forall$ quantifier where the rule can match zero occurrences of the subgraph). Then we require that there must exists a **Philosopher** node (dashed blue, attached to the $\exists$ node via @ edge which in turn is nested inside the $\forall^{>0}$ quantifier), which is in fact the same as the reader **Philosopher** node (expressed with the $=$ edge) and where its hunger value is equal to zero.

Since one of the **Philosopher** nodes is an eraser node, the matching philosophers get deleted upon rule application, modelling the collective leaving of the table.

3.2.4 Rule Priorities

In a graph state where more than one rule is applicable, it may be desirable to force a certain order when exploring the state-space. GROOVE allows controlling the order of rule applications. Using simple rule priorities—integer values associated with a rule—a system applies rules with higher priorities before rules with lower priorities. For example, if our philosophers do not like to eat alone, we could assign the rule pick_up a higher priority than the rule eat, which means that as long as there are philosophers which are able to pick up their forks, they do so, and once no additional philosopher can pick up its forks anymore are the ones currently having the forks allowed to eat.

GROOVE also provides more advanced mechanisms for controlling rule applications. In particular, control programs allow specifying complex expressions with conditionals, loops, choices, and other control flow mechanisms. Since we do not use control programs in this work, we do not discuss them here. Instead, we refer the interested reader to [20].
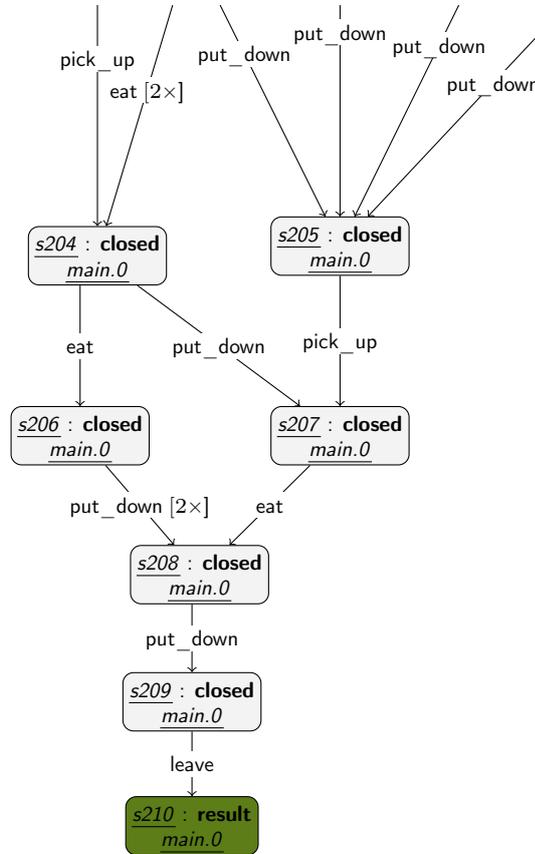
Figure 3.9: Excerpt from LTS of the dining philosophers model. After the last philosopher has put down his forks (transition from $s208$ to $s209$), all philosophers leave and the simulation has reached a final state ($s210$) where no more rule applications are possible.

3.2.5 Verification by Model Checking

Now that we have modelled the dining philosophers problem, we can generate the state-space and inspect it. GROOVE has many options for state-space exploration, for example it can do breadth-first search, depth-first search, random linear exploration, and other exploration types. A state is called *final*, if no modifying rule (a rule with erasers or creators) is applicable anymore. If we want to show that the dining philosophers example always results in the philosophers leaving the table (i.e. applying the `leave` rule before being in a final state), we could do this by generating the full state-space and inspecting paths and rule applications in the LTS, which is a graph where nodes represent states and edge labels denote which rule leads to the outgoing state. An excerpt of an LTS generated with our example can be seen in Figure 3.9.

Obviously, inspecting an LTS by hand is not feasible for larger state-spaces. Fortunately, we can specify LTL and CTL formulae in GROOVE. In our example, we could verify that all executions end up with the `leave` rule being evaluated with

the LTL formula `F leave`, expressing that, starting from the initial configuration, eventually the rule `leave` is applied. Since one can use arbitrary rules in LTL and CTL formulae, we can create rules capturing certain properties, such as a generic deadlock, of a system state and include them in the formulae.

3.3 Related Work

Our focus on GTS is limited to the theory relevant to GROOVE. An introduction to the algebraic approach, in particular to the DPO approach, can be found in [17], and a thorough discussion of the algebraic approach in [5].

While we focus on the GROOVE features relevant to this thesis here, the GROOVE User Manual [20] provides a more detailed description of GROOVE features. A set of best practices when working with GROOVE is presented in [25].

Several papers [19, 18] discussing GROOVE have been published, and GROOVE is compared to other simulation and model checking tools in [6].

Chapter 4

Towards a Concurrency Model for SCOOP

As we have seen in Chapter 2, SCOOP is a rich programming model that provides a framework for concurrent programming and is equipped with advanced object-oriented features. While this is great from a user perspective, it also makes modelling of the complete language a difficult task. To conquer this difficulty, we first isolate concurrency related features from SCOOP to obtain a subset of the model called CoreSCOOP. This subset of SCOOP is formalised by the *Concurrent Processor Model* (CPM) [8], a GTS based formal model. Thanks to the modular and extensible nature of the model, more features from SCOOP are added and eventually become CPM+OO as presented in Chapter 5. In this chapter, an overview of CoreSCOOP and a detailed description of CPM and its primary implementation in GROOVE is given. In the next Chapter, we then present CPM+OO, which adds object-oriented features from SCOOP to CPM.

4.1   CoreSCOOP

We define a small subset of SCOOP called CoreSCOOP. In this subset, only basic object-oriented features exist. There are only three kinds of data: integers, Booleans, and references to processors. A processor can execute a simple method with statements that modify local data—such as assigning a sum of two local integers to another local variable—as well as asynchronous commands and synchronous queries, where the target must be a different processor. A method can not call other methods on the same processor as there are no local calls. Simple method calls can be simulated by inlining the called method, but this does obviously not work for recursive calls.

The main part of CoreSCOOP—handling queries and commands—remains as in SCOOP. To enqueue a feature request in some processors request queue, one has to first obtain a lock to the queue of the target processor. While SCOOP handles locking implicitly by requiring separate targets to be controlled, CoreSCOOP handles locking explicitly, and locking can occur at any place in a method.

CPM is a formal model for CoreSCOOP and follows the specification in [13]. In the next section, we present this formalisation and its primary implementation as a GTS in GROOVE.
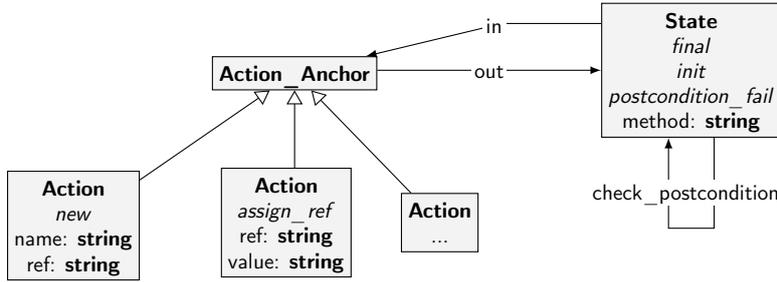
Figure 4.1: Control flow type graph.

4.2 CPM

CPM is a GTS modelling the behaviour of CoreSCOOP. It allows simulating configurations with a number of processors, each one performing computations on integer, Boolean, and reference values. We discuss the system, in particular the production rules involved, in detail in the following sections, by separating concerns in the following groups: control flow, system state, and queries and operations. We then discuss how the rules are prioritized to achieve the desired behaviour.

4.2.1 Control Flow

In a CPM start graph, methods are stored as control flow subgraphs. Figure 4.1 shows the relevant subset of the type graph of CPM. Methods start with an initial state (nodes of type **State** with the *init* flag), which is labelled with the method name. From state nodes, outgoing edges (labelled in) lead to action nodes, which in turn have an edge (labelled out) leading to state nodes. A final state node does not have an outgoing edge and denotes the end of the method. Action nodes contain information about the type of action (e.g. assignment, processor creation, locking) and additional data relevant to the action (e.g. a query target or command parameters).

There are a number of relevant rules, namely the following.

action_... The CPM GTS contains a number of action nodes. These nodes represent atomic units of work such as assignments, locking, commands, and so on, and can be compared to statements in a SCOOP program (although there are explicit locking actions that do not have a counterpart in SCOOP, where locking is done implicitly). Actions have the lowest priority, thus are applied when other rules, in particular scheduling rules for queues, can not be applied anymore. The following actions exists in CPM:

- action_Assign_... group: These rules perform an assignment operation. The assignment operation has been split in a number of sub-rules in order to keep individual rules simple, as there are a number of different scenarios for assignments: assignments of references and primitive data, void assignments, assignments to fresh variables and used variables, assignments to the special `Result` (return) value.

- action_Command: This action performs a command, and is shown in Figure 4.2. Since commands are always asynchronous (and therefore executed on a different processor), a **Queue_Item** is created and put on the processor that handles the target node. This enables queue management rules to be applied, which then eventually result in the target processor executing this particular request. Since the action_Command rule advances the in_method edge, the calling processor can continue execution (once action rules are enabled again).

- action_Lock_1 and action_Lock_2: These actions acquire locks for one or two processors respectively, cf. Figure 4.3 for an illustration of the latter. Embargo nodes prevent the rules from being applied if a processor is already locked by another one.

- action_Unlock: The counterpart to the lock actions consists of a single rule, as unlocking multiple locks does not have to happen atomically.

- action_Unlock_Creator: When creating new processors, the created processor is locked by the creating processor. By convention, the next action of the creator is a lock action of the created processor. As a result, the creating processor has to wait until the creation procedure, which contains a Unlock_Creator action at the end, removes this lock. This mechanism simulates the behaviour of SCOOP, where creation procedures—even for separate objects—are executed sequentially.

- action_New_Attached and action_New_Void: These rules create a new processor and point the designated reference variable to the newly created processor. Again, this task has been split into two separate rules for readability reasons and to avoid excessive usage of quantifier nodes.

- action_Query: As opposed to the command action, the query rule only binds the result of the executed query to the target (by assigning the result to the **Data_Var** matching the store_to edge, see Figure 4.4). The Queue_Items are instead created by other rules (e.g. bexp_Query (Figure 4.6) for Boolean queries, which are discussed in the system state section).

- action_Test: This rule performs a Boolean test by advancing only if the evaluated expression is true. The preceding state node has in certain situations two in edges, each pointing to a test action node, where one action node points to a Boolean expression, and the other one to its negation, as illustrated in Figure 4.5. This implements an if-else branching mechanism, and guarantees that the processor can make progress.

- action_TestPostcondition: In case there is a configuration node that denotes that we want to check postconditions, this rule is applied when a processor is in a state preceding an action node with the test_postcondition flag. The rule matches if the test result evaluates to true, and puts the processor in a final state.

config_CheckPostcondition In case there is a **Configuration** node with the *check_postconditions* flag, postconditions will be checked. To do so, this rule follows a final state along the *check_postcondition* edge to another
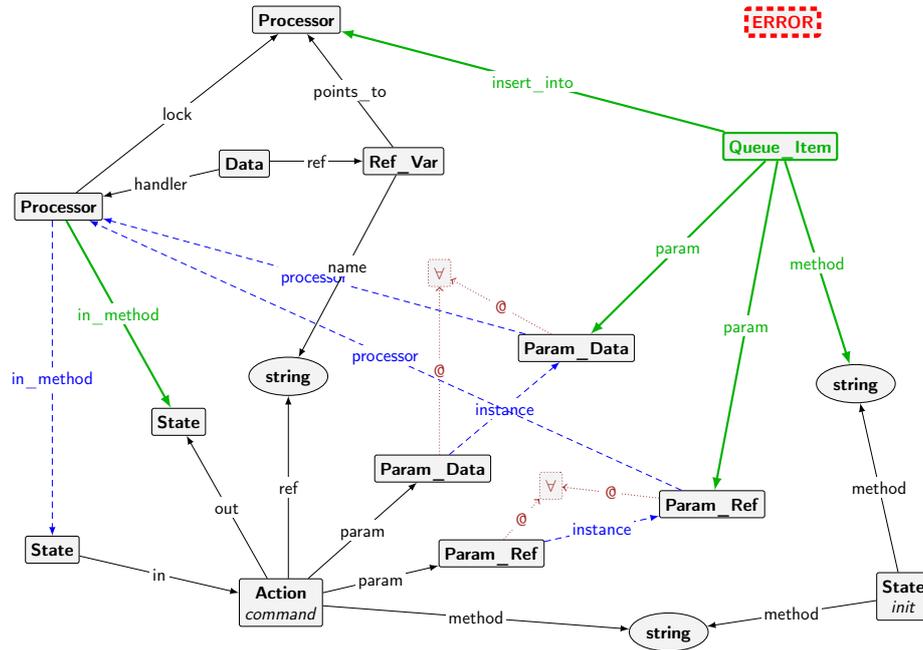
Figure 4.2: action_Command rule.

state node by redirecting the in_method edge of the relevant processor. Postconditions are the only situation where a state is followed directly by another state. The rule does not match if the configuration node is not present, providing an intuitive way to enable or disable postcondition checking.

4.2.2 System State

The system state is concerned with processors, queue management, and handling of data. The relevant type graph is shown in Figure 4.7.

Processors are at the core of CPM states. During their lifetime, they are either handling requests or they are idle. In the first case, they are executing a method at a certain position, denoted by the in_method edge. When requests are made by other processors, a **Queue_Item** is created which has a insert_into edge to the target processor, as can be seen in the rules action_Command (Figure 4.2) and bexp_Query (Figure 4.6). Once a queue item is created, a number of rules come into action that are responsible for queue management, namely the following:

queue_Insert_EmptyBusy and queue_Insert_NotEmpty These rules can be applied when a queue request has been made (with an insert_into edge), and their effect is to simply put the item at the end of the queue.

queue_Remove_ParamRef and queue_Remove_ParamData Nodes representing parameters are attached to the queue item upon creating it. These two rules prepare the call by removing the connection between queue item
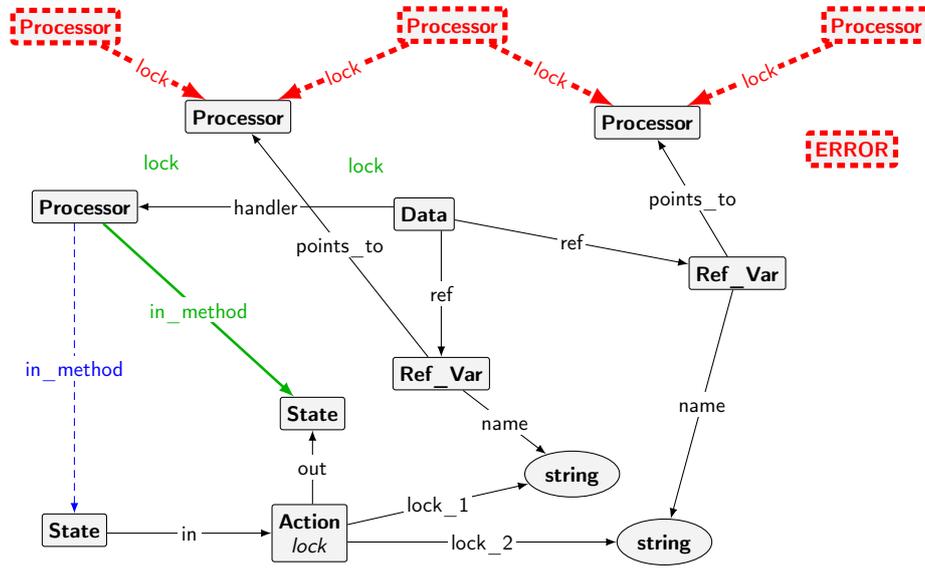
Figure 4.3: action_Lock_2 rule.
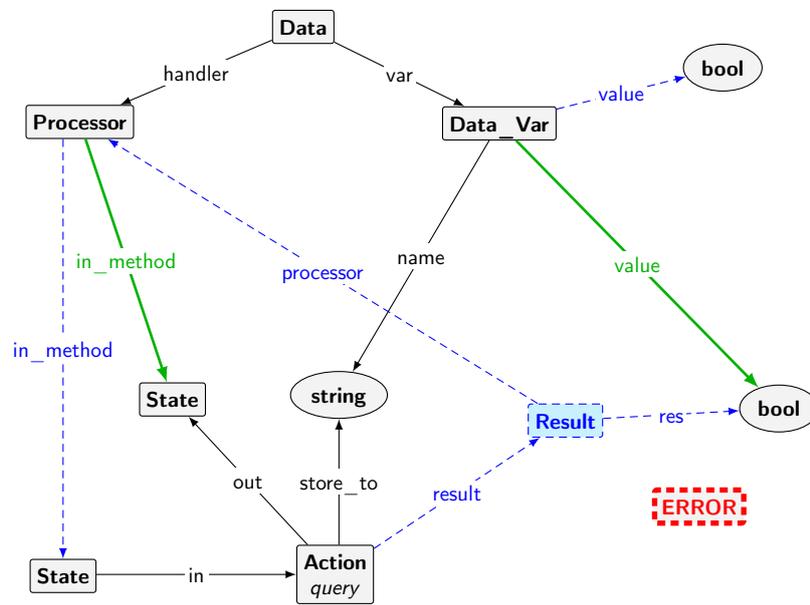
Figure 4.4: action_Query rule.
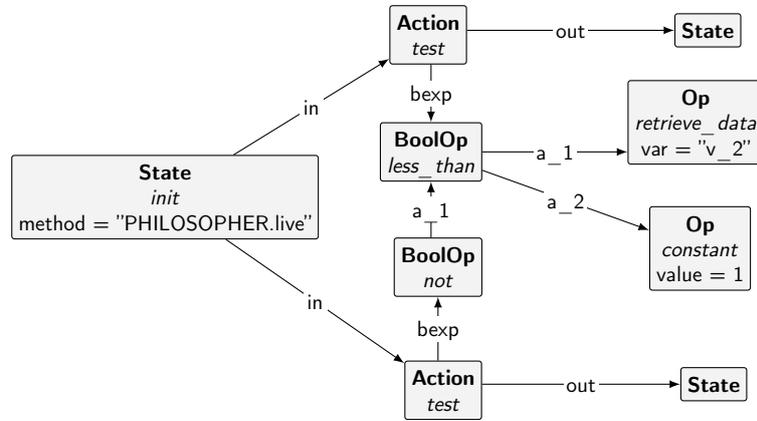
Figure 4.5: Excerpt from a start graph illustrating usage of two action nodes to provide an if-else construct.
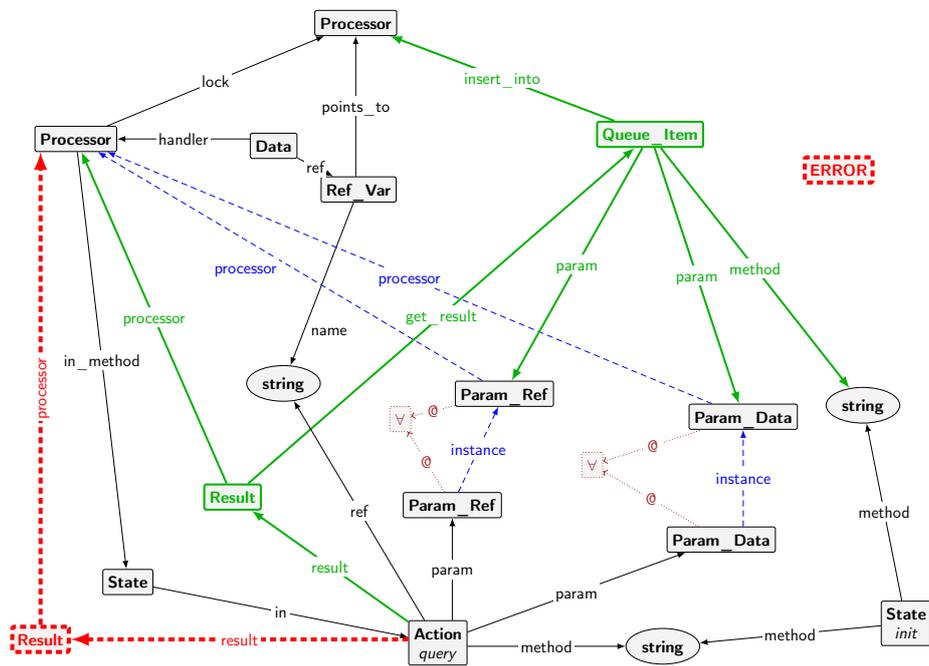
Figure 4.6: `bexp_Query` rule.
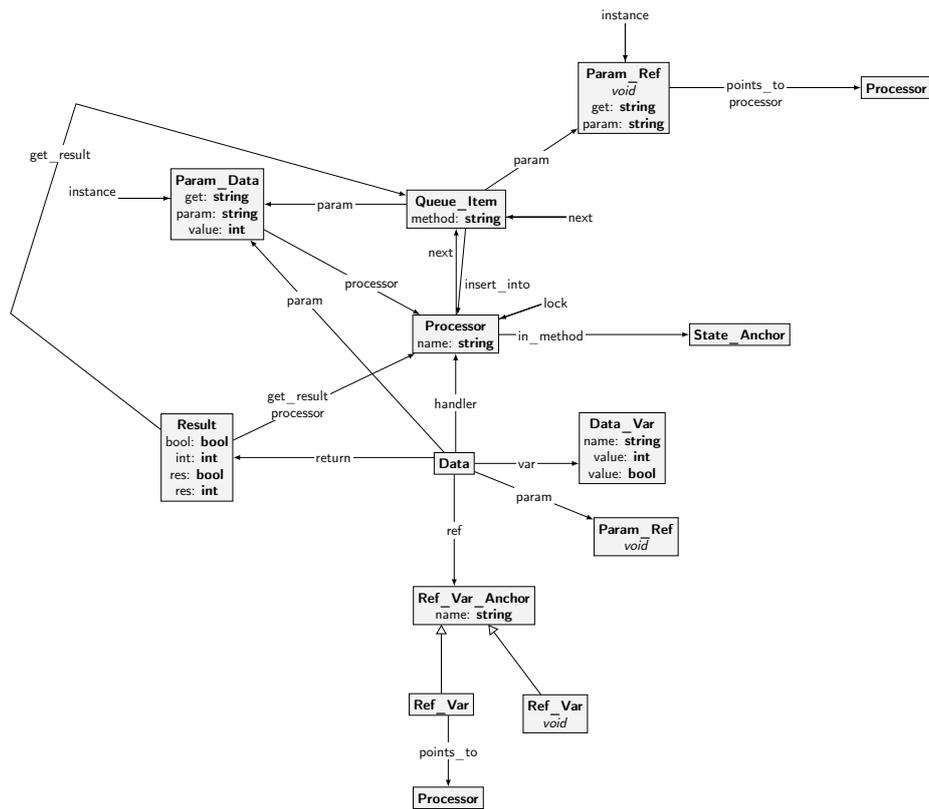
Figure 4.7: Type graph of the system state. Note that self edges are rendered by an arrow that leads from a label to a node (an example is the instance edge of the **Param_Ref** node). We render self-edges in this manner throughout this thesis.
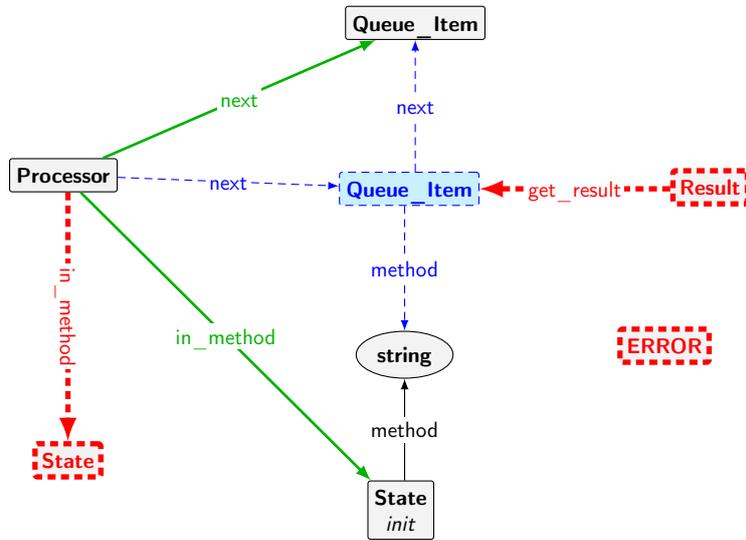
Figure 4.8: queue_Remove_Command_MultipleQueued rule.

and parameter node, and attaching the parameter to the processor's data node. The next rules will handle the remaining part of the queue item.

queue_Remove_... The remaining four rules in the queue_Remove group remove a query or a command request from the top of the queue and activate the processor to start execution at the given method. There are two rules for the case with one item on the queue and two rules for the case with more items on the queue.

Figure 4.8 shows the rule queue_Remove_Command_MultipleQueued, which handles the case of multiple queue items and the top item being a command request.

4.2.3 Queries and Other Operations

While there exist a query flag for action nodes, the rule that advances over an action node only does so after the query has been evaluated and is essentially an assignment operation where the right-hand side happens to be a query. Similarly, other assignment operations also contain right-hand sides that need to be evaluated before the assignment can be performed. For example, in the assignment r_1 := r_2, the reference on the right-hand side must first be fetched. The group of rules in this subsection handle this, and they have higher priorities than the action rules in order to make sure that whenever an action requires arguments, they are fetched first. The type graph for operations, shown in Figure 4.9, contains the operation types. Since the operation types are encoded using flags, it would be possible to have, for example, an **Op** node with both the *constant* and the *add* flag, as it is not possible to force having exactly one flag. By convention, we do not support multiple flags for such a node. We set up the type graph to reflect how the various node types are intended to be used.
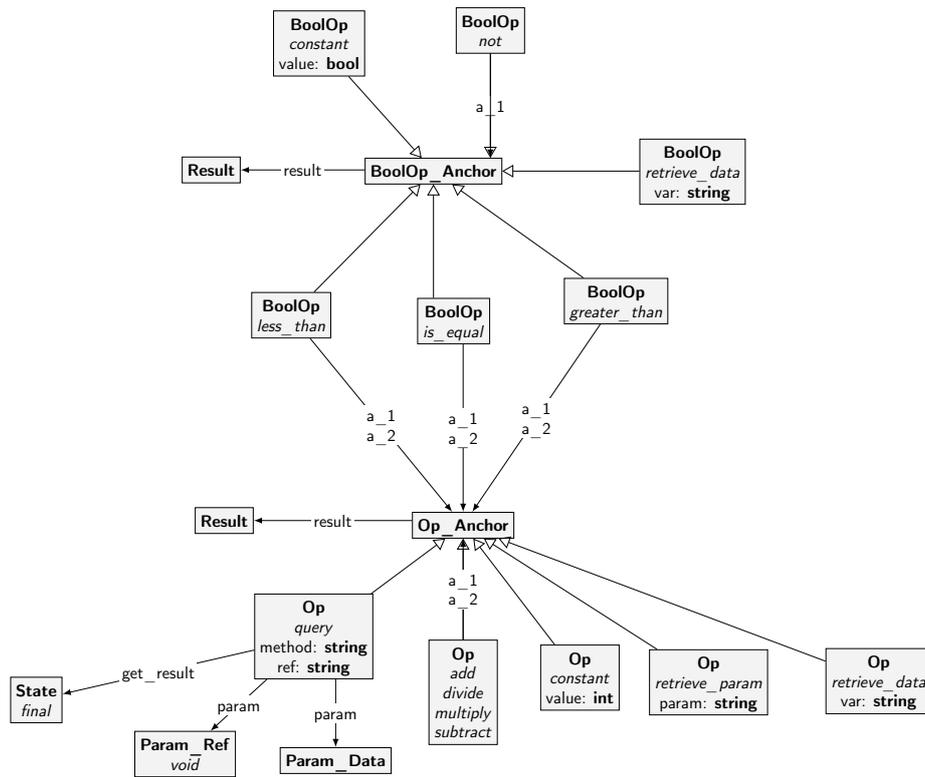
Figure 4.9: Type graph of operations and queries.

Queries do not appear as action nodes themselves. Instead, they are attached to an assignment action. Similarly, integer and Boolean operations are not targets either, as they appear in either complex expressions or on the right-hand side of an assignment. The relevant rules are the following:

aexp_... Arithmetic expression rules evaluate integer expressions, by creating a **Result** nodes and attaching them to **Op** nodes. The following rules exists for arithmetic expressions:

- aexp_constant: Creates a result with the value specified by the operation itself.
- aexp_RetrieveParam: Fetches a parameter from the data handled by the current processor.
- aexp_RetrieveData: Retrieves an integer data value from the current processor.
- aexp_Multiply, aexp_Divide, aexp_Add, aexp_Subtract: Evaluates the binary operation and creates a result node.

bexp_... Analogously to the arithmetic expression rules, Boolean expression rules evaluate Boolean expressions. The following rules exist in CPM:

- bexp_constant, bexp_RetrieveData: Analogous to the arithmetic expression variants.
- bexp_GreaterThan, bexp_LessThan, bexp_IsEqual, bexp_not: Evaluates the binary operation and creates a result node with a Boolean result.

bexp_Query The rule for Boolean queries creates a **Queue_Item** which will be inserted into the queue of the target processor, as illustrated in Figure 4.6. It is similar to the action_Command rule. The target processor will execute the request and once the result is available, it can be matched by the action_Query rule.

getparam_Ref_... This group consists of rules for fetching method parameters for command actions. They perform the step of looking up the value of a reference or data variable and create a **Param_Data** instance, as illustrated in Figure 4.10 for the integer case.

Queries and operations often require intermediate nodes. For example, we attach a **Result** nodes to an **Op** after evaluating it. Once the processor has used this result and moved past the state where it was required, we can safely remove it in order to keep the graph clutter-free. The following rules clean the graph in various situations.

cleanup_RemoveParamRef_Attached

cleanup_RemoveParamRef_Void

cleanup_DiscardParamData Once the processor is in a final state of a method, parameters are not required any more and are removed by these rules. In fact, we must remove them as otherwise the system may misbehave in subsequent method calls (e.g. if the next call has the same parameter names, two nodes of the same parameter exist and rules that match it have two possible applications).

Figure 4.10: getparam_Data rule.

cleanup_exp_DiscardResults_Op

cleanup_exp_DiscardResults_BoolOp Once a processor moves past an action
that has an **Op** or a **BoolOp** node attached, the corresponding result nodes
are not required any more and are removed by these rules.

cleanup_FinalState_BoolQuery and **cleanup_FinalState** These rules are ap-
plied when a processor reaches the final state of a query or command
respectively. The in_method edge is deleted, as well as associated edges
and nodes related to the result value.

4.2.4 Rule Priorities

As mentioned earlier, the rules in CPM are not applied with the same priorities.
This has various reasons. First of all, it enables control on what rules are applied
in which situations. For example, cleanup rules have priorities such that they are
performed before new actions are performed, ensuring that no "leftover" nodes
stay in the graph (e.g. parameter instances from earlier commands and queries).
If the cleanup rules do not have higher priorities, the graph may end up in a
state where an action rule has multiple matches, in particular matches with old
and invalid instance nodes.

Another advantage of rule priorities is that they can be used to attack the
state-space explosion problem. By assigning fine grained priorities to the rules
that do not influence the modelled behaviour, we reduce the possible interleavings
in an execution. For example, it does not matter in which order cleanup rules
are applied, as once all matching cleanup rules are applied, the system always
returns to the same state. By assigning each cleanup rule a unique priority value
and thus forcing a fixed order, these local interleaving scenarios are eliminated.

Of course one has to pay attention to which rules can have different priorities
and which ones need to have the same priorities. Action rules generally should
have the same priorities, since these nodes can create queue items and we are
interested in the interleavings with unique queue item sequences.

Our approach is in line with Zambons and Rensinks paper [25] on best
practices in GROOVE, where they suggest to use some form of rule scheduling
whenever possible. While they mention that "the use of control programs is
usually preferred over priorities", we think that priorities are sufficient and easier
to maintain in our case.

Table 4.1 shows a list of all rules in CPM and their priorities.

4.3 Dining Philosophers

4.3.1 Start Graph

This section revisits our running example of the dining philosophers by showing
CPM in action. A dining philosophers start configuration for CPM is shown in
Figures 4.11 on page 36 and 4.12 on page 37 (the graph has been split up for
readability, but both make up a single graph and are represented in GROOVE as
such).

Most of the nodes in Figure 4.11 belong to the control flow graph of APPLI-
CATION.make, which is the root procedure in this example. It roughly translates

Rule	Priority
error_Deadlock_2Proc_DiffState	100
error_Deadlock_2Proc_SameState	
error_DivideByZero	
error_NoSelfRef	
error_PostconditionFail	
error_VoidCall	
config_CheckPostcondition	60
cleanup_RemoveParamRef_Attached	50
cleanup_RemoveParamRef_Void	49
cleanup_DiscardParamData	48
cleanup_exp_DiscardResults_Op	47
cleanup_exp_DiscardResults_BoolOp	46
getparam_Ref_Attached	44
getparam_Ref_Void	43
aexp_constant	41
bexp_constant	40
aexp_RetrieveParam	39
aexp_RetrieveData	38
bexp_RetrieveData	37
bexp_Query	36
aexp_Multiply	35
aexp_Divide	34
aexp_Add	33
aexp_Subtract	32
bexp_GreaterThan	31
bexp_LessThan	30
bexp_IsEqual	29
bexp_not	28
queue_Insert_EmptyBusy	20
queue_Insert_NotEmpty	19
queue_Remove_ParamRef	18
queue_Remove_ParamData	17
queue_Remove_BoolQuery_MultipleQueued	16
queue_Remove_BoolQuery_SingleQueued	15
queue_Remove_Command_MultipleQueued	14
queue_Remove_Command_SingleQueued	13
cleanup_FinalState_BoolQuery	5
cleanup_FinalState	4
action_TestPostcondition	2
action_Assign	0
action_AssignRef_Param_Attached	
action_AssignRef_Param_SameTarget	
action_AssignRef_Param_Void	
action_AssignRef_Ref_Attached	
action_AssignRef_Ref_SameTarget	
action_AssignRef_Ref_Void	
action_Command	
action_Lock_1	
action_Lock_2	
action_New_Attached	
action_New_Void	
action_Query	
action_Test	
action_Unlock	
action_Unlock_Creator	

Table 4.1: Rule priorities. Note that an empty priority means that the rule has the same priority as the one above it, e.g. all error rules have priority 100.

to the code in Listing 4.1, with a simple loop that instantiates forks and philosophers and connects them accordingly. In addition to the method graph, there is also a **Processor** node. It is the handler of its data, which consists of a number of reference and integer variables. Note that the variables have generic names (such as `v_1` for Boolean and integer data and `r_1` for references), and that there is a mapping from CPM variable names to the variable names of the code in the listing.

```
1  class APPLICATION
2    feature
3      make
4        do
5          -- variable mappings:
6          --   v_1: i
7          --   v_2: philosopher_count
8          --   v_3: round_count
9          --   r_1: first_fork
10         --   r_2: left_fork
11         --   r_3: right_fork
12         --   r_4: a_philosopher
13         from
14           i := 1
15           create first_fork.make
16         until
17           i > philosopher_count
18         loop
19           if i < philosopher_count then
20             create right_fork.make
21           else
22             right_fork := first_fork
23           end
24           create a_philosopher.make (i, left_fork, right_fork,
                  round_count)
25           lock (a_philosopher)
26           a_philosopher.live
27           unlock (a_philosopher)
28           left_fork := right_fork
29           i := i + 1
30         end
31
32    i, round_count: INTEGER
33    first_fork, left_fork, right_fork: separate FORK
34    a_philosopher: separate_PHILOSOPHER
35  end
36
37  class FORK
38    feature
39      make
40        do
41        end
42  end
43
44  class PHILOSOPHER
45    feature
46      make (philosopher: INTEGER;
47            left, right: separate FORK;
48            round_count: INTEGER)
49        do
50          -- variable mappings:
51          -- p_1: philosopher
52          -- p_2: left
```

```
53          -- p_3: right
54          -- p_4: round_count
55          -- v_1: id
56          -- v_2: times_to_eat
57          -- r_1: left_fork
58          -- r_2: right_fork
59          id := philosopher
60          left_fork := left
61          right_fork := right
62          times_to_eat := round_count
63        end
64
65    live
66        do
67          lock (left_fork, right_fork)
68          -- eat
69          unlock (left_fork)
70          unlock (right_fork)
71          times_to_eat := times_to_eat - 1
72        end
73
74    left_fork, right_fork: separate FORK
75    id, times_to_eat: INTEGER
76 end
```

Listing 4.1: `APPLICATION` class for the dining philosophers.

Figure 4.12 shows the control flow graphs for forks and philosophers. Processors representing forks do not execute any code after their creation procedure. They are created, the *unlock_creator* action is performed, and afterwards they just exist in the system, but do not execute more requests (as no other processor ever performs a command or query on a fork). In the make method of the philosopher, the object is initialized by assigning the parameters to the processor's reference variables and data variables. Finally, the subgraph representing the live method is traversed to perform the main loop of the philosophers. Most notably, this subgraph contains actions to lock (representing atomically acquiring the forks and eating) and unlock the fork processors.

4.3.2 Rule Applications

With the start graph presented in the previous section, we can now inspect the behaviour of CPM. With the GROOVE Simulator, it is possible to follow the state-space exploration visually. Applicable rules are pointed out to the user and the part of the graph that matches is highlighted. Figure 4.13 on page 38 shows the program right before the first creation procedure (a command) is performed. At this point, the rule action_Command (see Figure 4.2 on page 23) is the only rule that has a match and is highlighted in green. After applying the rule, the graph looks as depicted in Figure 4.14 on page 39.

Of course, going through rules using the Simulator is a rather tedious task that may be a useful tool for developing, testing, and debugging such systems, but it is not for verification purposes. Fortunately, as we have seen in Chapter 3, GROOVE provides utilities to verify for LTL and CTL formulae. This is where the error rules come into play. To verify whether the program deadlocks, one can simply try to find a counterexample for the formula

! F (error_deadlock_2Proc_DiffStates | error_Deadlock_2Proc_SameState),

which states that, starting from the start graph state, there is no future state where either one of the mentioned rules matches.

Figure 4.11: Subgraph of the dining philosophers start graph in CPM, consisting of the initial processor starting at the root method APPLICATION.make.
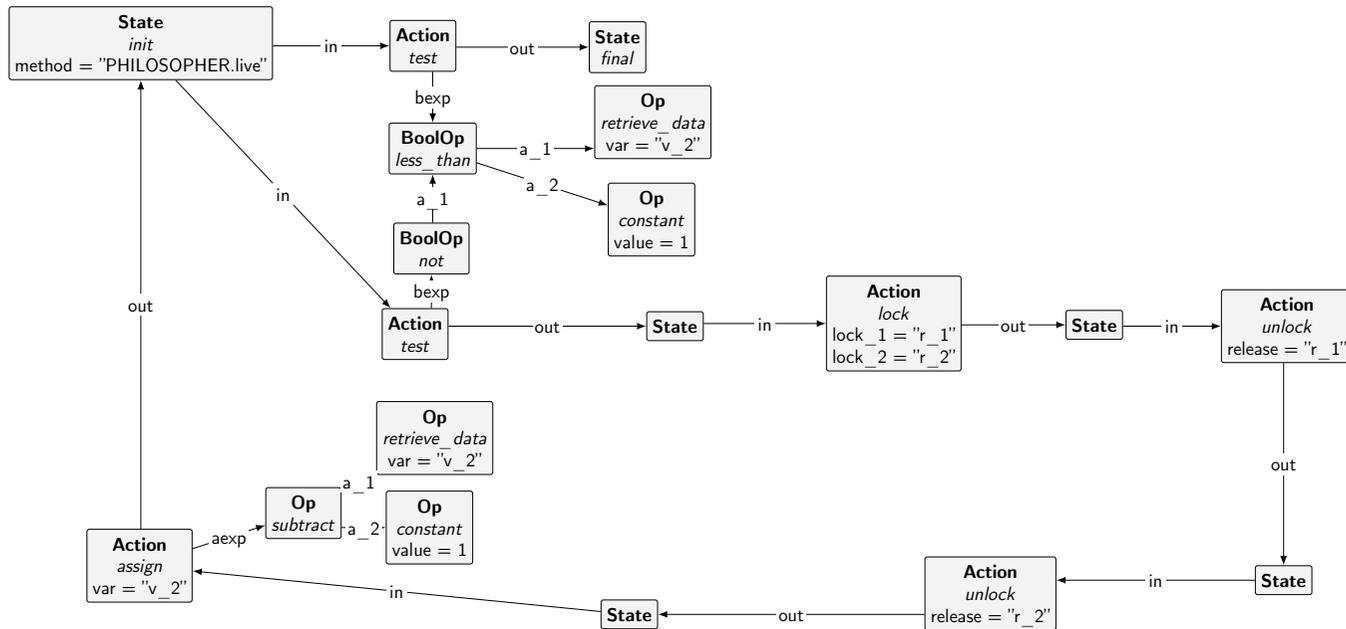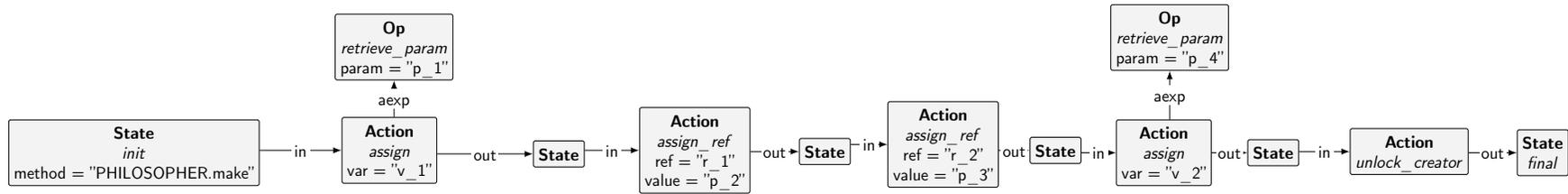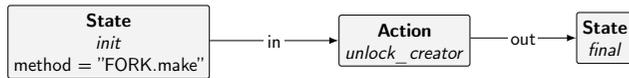
Figure 4.12: Various features of the PHILOSOPHER start graph.
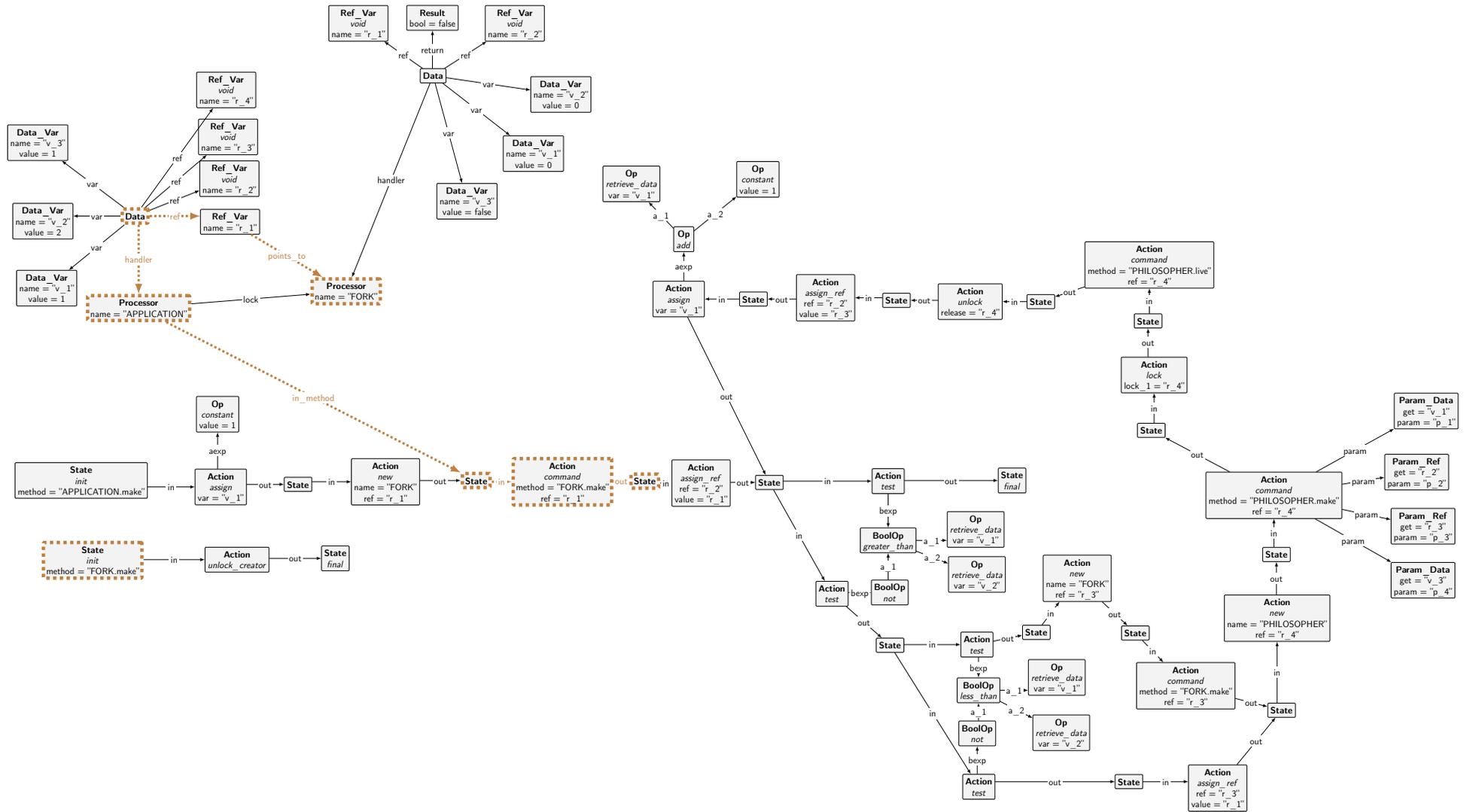
Figure 4.13: Configuration where the processor is about to execute a command action. The match is highlighted with dashed lines.
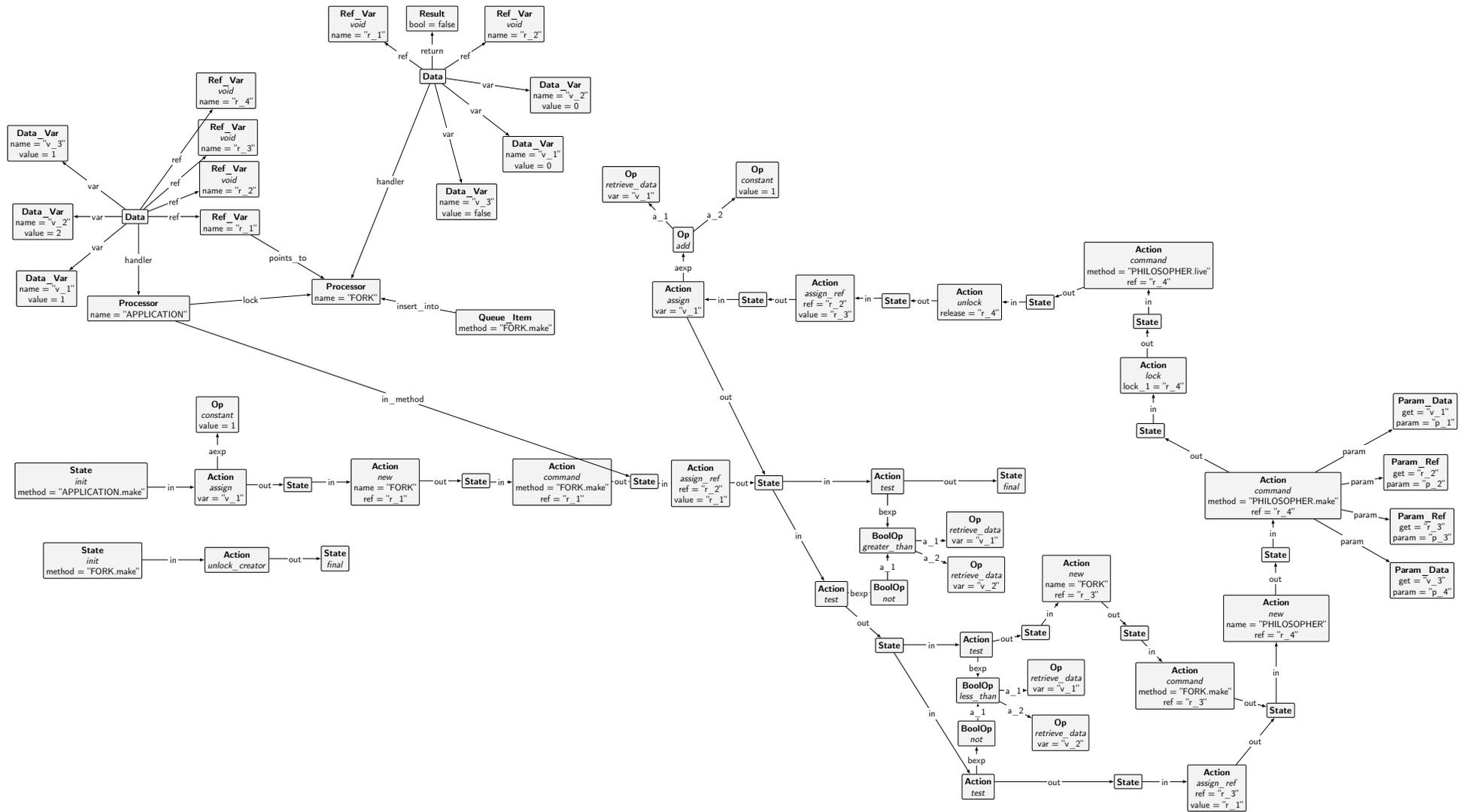
Figure 4.14: Configuration right after an application of the command action rule. A new request has been created and is about to be inserted into the request queue of the target processor.

Chapter 5

CPM+OO: An Extension for Objects

CPM *with Object Orientation* (CPM+OO) builds on top of CPM, and aims to bring back object-oriented concepts that have been intentionally left out from CoreSCOOP and CPM. While CPM focuses on the concurrency aspects of SCOOP, it can be difficult to map real-world SCOOP programs, with processors potentially handling multiple objects, non-separate calls within routine bodies, and other SCOOP features that are not directly modelled in CPM. These enhancements allow a more direct mapping of SCOOP programs to the graph model and clear the path for an automatic translation tool from SCOOP programs to CPM+OO (cf. Chapter 6).

   In this Chapter, we discuss various extensions made to the CPM model. We explain how the changes affect the behaviour of the system and make informal arguments for the preservation of soundness and completeness.

5.1   Type Graph Overview

We start by giving an overview over the changes of various parts of the type graph, which may seem overwhelming at first, as the type graph has changed significantly between CPM and CPM+OO. The goal of this section is not to provide a complete description, but instead relate the updated type graph to the various added concepts, which will then be explained in detail in subsequent sections.

5.1.1   Processors, Frames, and Objects

To model local calls and non-separate objects, we introduce the notion of stack frames and object instances to the model. A subgraph of the CPM+OO type graph with processor and object related nodes can be seen in Figure 5.1.
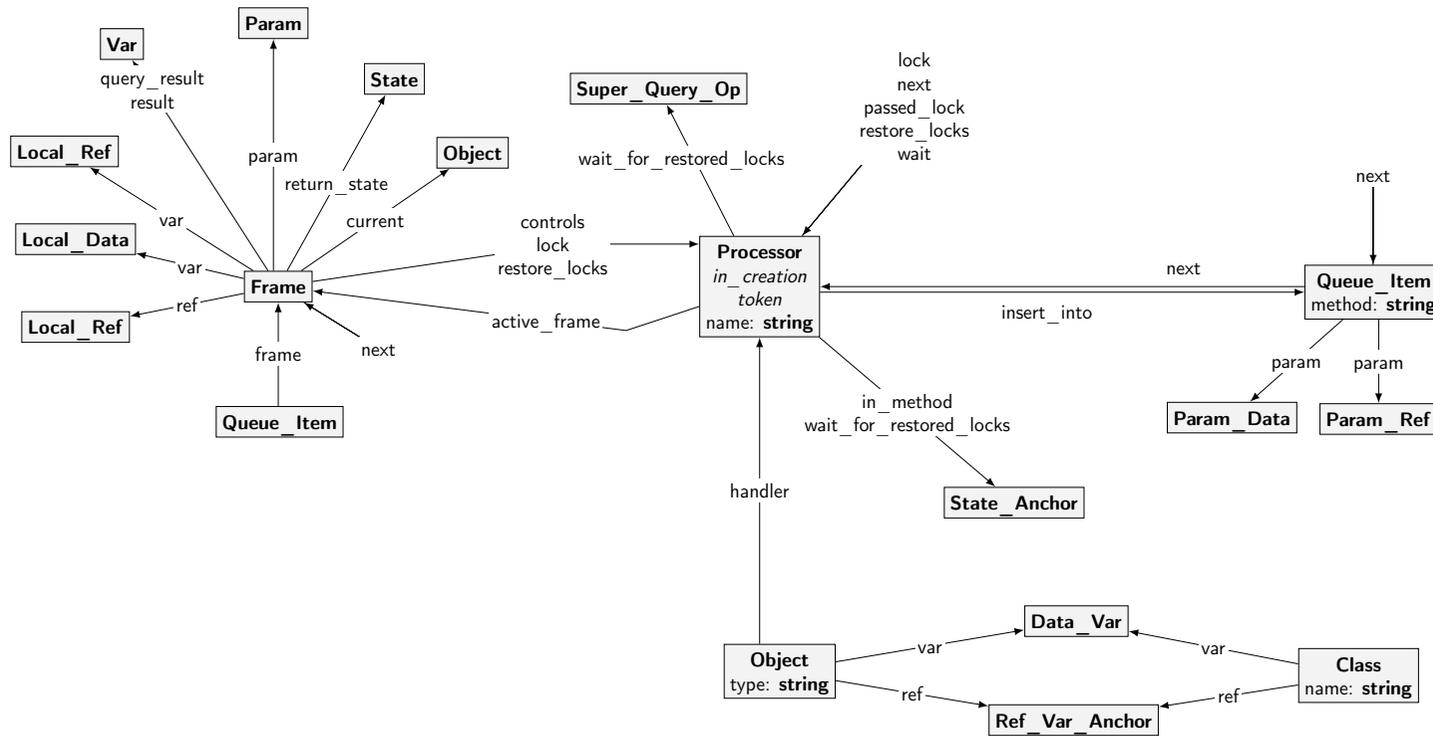
Figure 5.1: Type graph of processors and objects.

At the core of the type graph is the **Processor**. Tied to it is the **Queue_Item** with its parameters and method name that is intended to be executed.

The basic **Data** node has been replaced with the **Object** node by simple renaming (note that the type graph simply puts syntactic restrictions on the graphs, different semantics introduced in CPM+OO come only with the rule changes). A processor can now be the handler of multiple objects (whereas CPM restricted a processor to be the handler of exactly one data node).

Introducing nested routine calls brings the requirement for some call stack representation. This is achieved by introducing the **Frame** node type and its attached nodes. A processor that is executing a program always works with a current stack frame, which handles local variables, passed parameters, a reference to the `Current` object, and the state it needs to return to in case the current call is a nested call. When creating requests, a queue item has a stack frame attached, which contains passed parameters.

5.1.2 Variables, Parameters, and Results

The next group of related types is concerned with the representation and handling of variable declarations and bindings of values to parameters and query results. Figure 5.2 shows the relevant type graph.

Variable declarations are, as in CPM, divided into three different subtypes: reference, integer and Boolean variables. While integer and Boolean data variables did not change, reference variables now point to objects instead of processors. In addition, reference variables now have a flag denoting whether it is declared as separate or not.

Parameter nodes represent values passed to commands and queries. Parameters can be local variables (corresponding to variables in the `local` block) or attributes of the current object, as well as arbitrary expressions (**Param_Expr** with an expr edge to an operation node). Adding arbitrary expressions (as opposed to local variables and attributes only) as parameters allows representing complex expressions with a single action node in CPM+OO. This is not possible in CPM, where helper variables are required to simulate complex expressions.

5.1.3 Actions

The type graph for actions, shown in Figure 5.3, looks similar to the one in CPM. One important difference is that we use subtypes for the different kinds of actions as opposed to flags. This has the advantage that we can not have a single action node with multiple types, a property which can not be enforced in GROOVE using flags.

To support arbitrary expressions (e.g. `sum := a1.count + a2.count`) as parameters and operands, we replace a number of string attributes with edges to nodes of type **Super_Op** (and its subtypes). For example, the unlock action node **Action_Assign_Ref** is pointing to a **RefOp** node, which means that we can use arbitrary expressions on the right-hand side of the assignment.

The type **Action_Token** is a supertype of actions that are local to a processor, or, in the case of queries, potentially local to a processor. This is later used in a mechanism to mitigate the state-space explosion problem and is discussed in detail in Section 5.3.
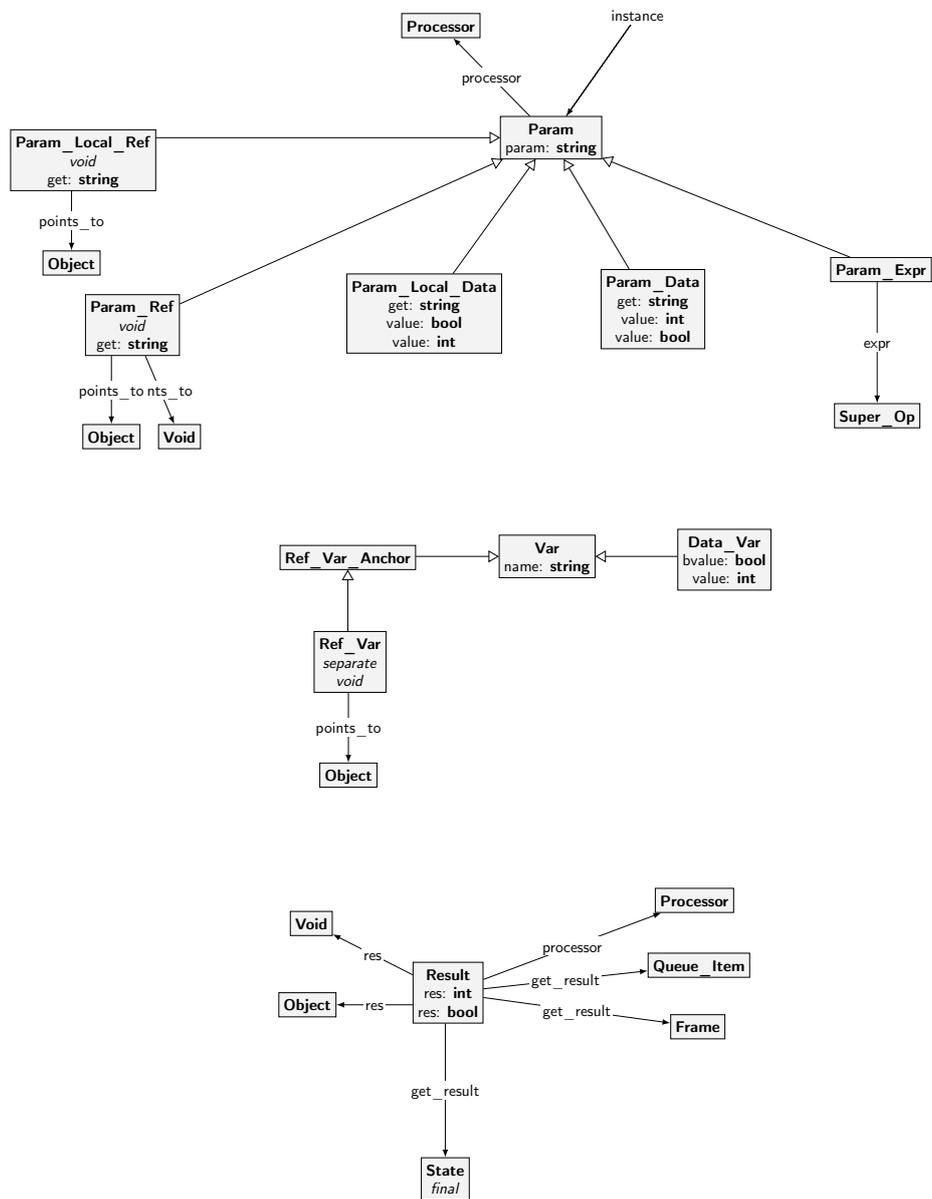
Figure 5.2: Type graph of variable, parameter, and result nodes.
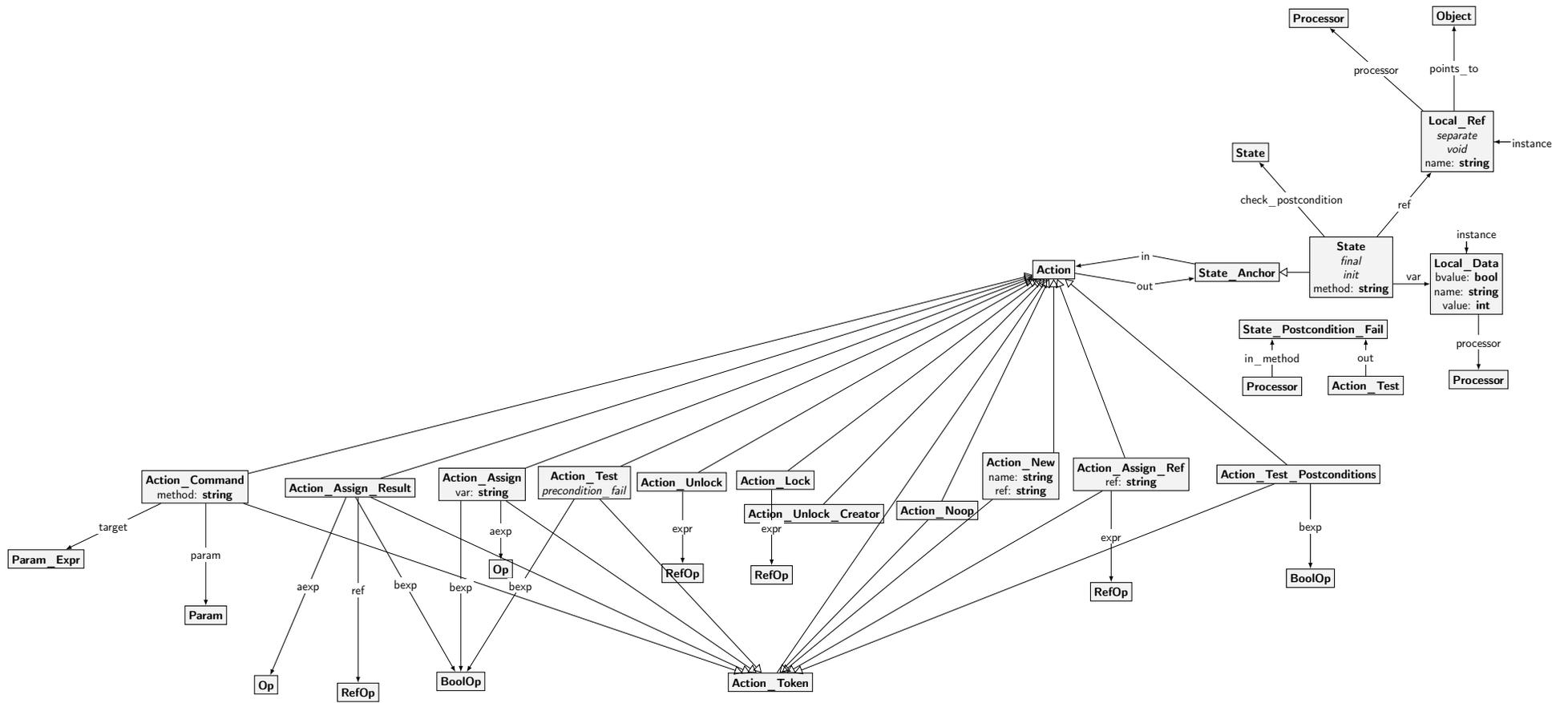
Figure 5.3: Type graph of actions.
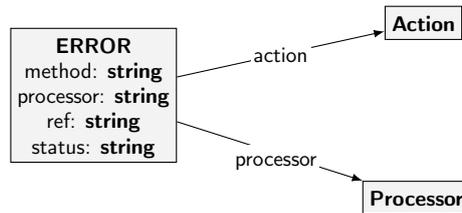
Figure 5.4: Type graph of errors.

5.1.4 Operations

The group of operation types (cf. Figure 5.6 on page 47) has undergone a number of changes during development of the CPM+OO model. Integer, Boolean, and reference operations (**Op**, **BoolOp**, and **RefOp** types) do now inherit from the same **Super_Op** supertype, which can be matched in certain rules to cover all kinds of operations. The different operations (e.g. "greater than" and "equals" operations) are represented as unique types, which is more restrictive than representing the operations with flags for one single node type. Other additions include types for the handling of local declarations as well as types for integer and reference type queries.

5.1.5 Errors

As in CPM, the **ERROR** type (Figure 5.4) is used for recording information about detected issues with the program. This includes both undesirable properties in the behaviour of the program (e.g. a deadlock situation) as well as invalid configurations (e.g. multiple handlers for a single object). The latter kind of error is used to aid the development and evolution of the model and is designed to catch bugs in the model itself, not errors in the behaviour of the modelled runtime.

Recording information in error nodes is useful for postprocessing. Since all rules have an **ERROR** embargo node, rules can only be applied as long as there is no error node. Once an error node is created, the system is in a final state. Our postprocessing tools can then simply go through all the final states and check whether there is an error node, and if so, generate output based on the context of the error.

5.1.6 Others

Figure 5.5 shows the remaining types in CPM+OO. They are used in the model as follows:

- **Reset_Token** and **Action_Executed_Indicator**: These two types are used for an optimisation technique that forces processors that are performing non-separate actions to advance as far as possible before yielding control to the next processor. We describe the state-space optimisation involving these types in detail in Section 5.3.

- **Configuration**: A node of this type can be included in a start graph to enable special behaviour of the model. In particular, one can add the

Figure 5.5: Miscellaneous types.

check_postconditions flag to enable postcondition checking.

- **Init**: The initialization type allows specifying the root class and procedure.

Figure 5.6: Type graph of operations.

5.2   Modelled SCOOP Features

5.2.1   Local and non-separate Calls

In CPM, calls are always performed by adding a new queue item to the request
queue of a remote processor. Local calls are not supported, instead one has to
perform inlining of the method bodies where local calls would occur. CPM+OO
instead provides mechanisms for local routine calls (i.e. where the target is the
current object) and non-separate calls (where the target may be a different
object, but is handled by the current processor).

   To achieve this, we use the call stack representation introduced in the type
graph, and introduce rules that handle non-separate calls. The corresponding
rules for separate calls are derived from the CPM rules that create feature requests
and are enhanced with the notion of a call stack and adapted to the object
representation.

   In the following, we first discuss separate calls. We show how we the CPM
rules to work with objects, frames, and other changes in the CPM+OO type graph.
We then present features that are missing inCPM, such as non-separate calls.

Separate Calls

The rules handling creation of feature requests, i.e. separate feature calls, are
similar to what we have seen in CPM (rules action_Command and bexp_Query,
see Figures 4.2 on page 23 and 4.3 on page 24). Figure 5.11 shows the rule
action_Command_separate from CPM+OO. While the rule is much larger than its
CPM counterpart, the semantic behaviour of the two do not differ much. There
are several reasons why the CPM+OO rule requires more nodes. First, CPM+OO
supports additional parameter nodes for a command (in particular, expressions,
local references, and local data), resulting in more pairs of parameters and
instances, but they follow the same structure as data and reference parameters in
CPM. Second, to the lower left, there is a construct nesting an ∃ quantifier inside
a ∀ quantifier. Since the parameter node is at the ∀ quantifier and its instance
at the ∃ quantifier, this expresses that the rule matches, if for all Param_Expr
nodes of the command, there exists an instance of type Param. This construct is
used to enforce that the rule is only applied once all parameter expressions are
evaluated (i.e. once instance nodes have been created). Once the requirements
are met, a queue item is created, similar to what the CPM rule does. But instead
of attaching parameters directly to the queue item, we create a **Frame** node,
representing the prepared stack frame which can be put on top of the frame stack
once the request is handled. Note that the action does not specify a reference
denoting the call target as a simple string any more, instead it points to a
parameter expression via the target edge which allows using complex expressions
as targets (e.g. `foo.get_counter ().count`).

   Once a processor has queue items attached, the scheduling rules for queues
are applied. These work analogously to the CPM counterparts. For example,
Figure 5.7 shows the rule queue_Remove_SingleQueued in CPM, while Figure 5.8
shows the corresponding rule from CPM+OO. In CPM+OO, we not only point the
processor to the routine that should be executed, but also set the active frame
edge to the frame attached to the request, thus providing information about the
routine arguments and the target object (this is needed since the processor may

Figure 5.7: Rule queue_Remove_Command_SingleQueued in CPM.

be handling multiple objects and needs to know to which one **Current** refers
to). Note that, since we attach certain information related to the request type to
the frame instead of directly attaching it to the request, CPM+OO requires only
two general queue_Remove_... rules (as opposed to different rules for queries
and commands in CPM).

After the execution of a request, several rules take care of cleaning up the
frame and handling of possible return values. For example, the rule cleanup_Fi-
nalState_Commands_Empty_Call_Stack, shown in Figure 5.9, shows how the
stack frame is deleted after a command when the processor becomes idle.

Non-separate Calls

Thanks to stack frames, we can now also simulate non-separate calls and local calls
(calls with target **Current**). These commands and queries, as per the formal
semantics, do not create a feature request that is enqueued in the processor's
request queue. Instead, they are directly (and sequentially) executed by the
calling processor. We stay with the command example, but the case for queries
is again similar. The rule action_Command_non-separate, shown in Figure 5.12,
handles command calls. Again, a stack frame is created. But instead of creating
a new request, the processor updates its current frame pointer to the newly
created frame, and starts executing the desired method body. The created frame
points to the current frame via next edge. In addition, the created frame also
includes an edge to the state after the command node, labelled return_state.
This allows returning to the correct position once the command finishes and the
stack frame is removed.

Once a procedure has been processed and the processor points to a final state

Figure 5.8: Rule queue_Remove_SingleQueued in CPM+OO.

Figure 5.9: Rule cleanup_FinalState_Command_Empty_Call_Stack.

Figure 5.10: Rule cleanup_FinalState_Command.

node, several high-priority scheduling rules may be applied, depending on the state of the call stack and the type of feature (command or query). These rules (which in fact are the same ones that handle the analogous case for separate features) start with the prefix cleanup_FinalState_... and pop the current frame from the frame stack. Figure 5.10 shows the cleanup_FinalState_Command rule that deletes a frame and instructs the processor to continue at the position after the call in the calling procedure, and resets the active frame edge to point to the original stack frame.

Figure 5.11: Rule action_Command_separate.
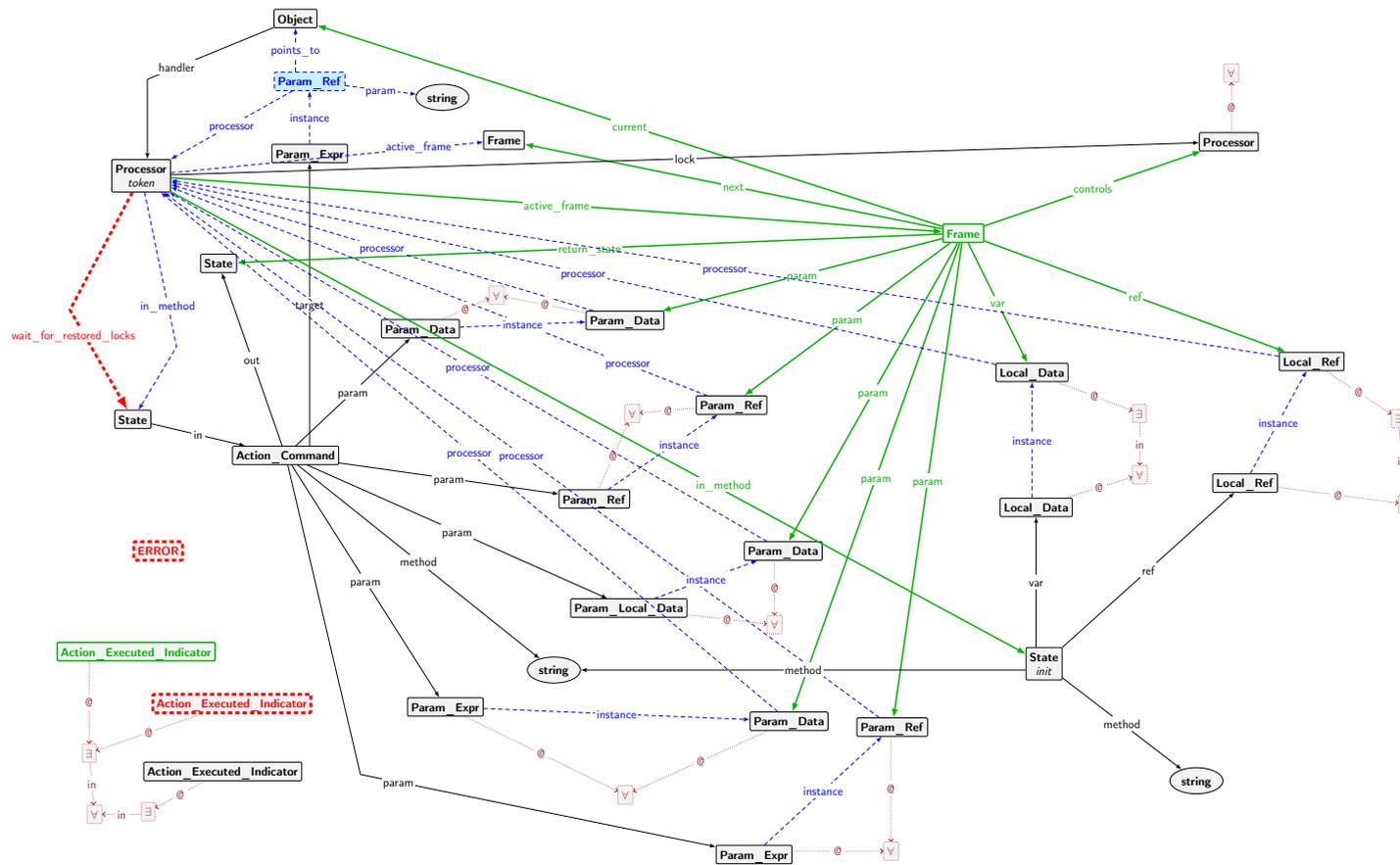
Figure 5.12: Rule action_Command_non-separate.
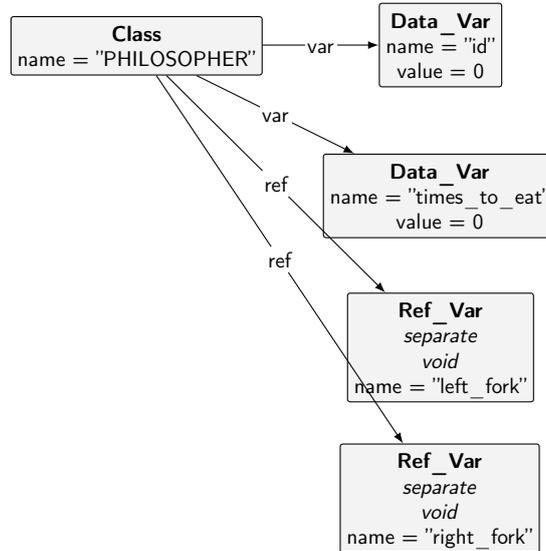
Figure 5.13: Object template for the PHILOSOPHER class.

5.2.2 Dynamic Object Creation and Variable Names

When creating processors and associated data nodes in CPM, the rules action_New_Void and action_New_Attached are applied to create a fixed number of reference and data variable nodes. To map a certain program to CPM, one has to first determine how many variables are required to represent the program and then adjust these rules to make sure that enough variables are available for the mapping. If a program involves several classes, CPM does not distinguish between them when creating data and handlers. Instead, all processors get the maximum number of data and reference variables.

To make direct translations of SCOOP programs easier and interpreting generated start graphs more intuitive, we introduce variable names in CPM+OO that can be directly mapped to the names in the source code. To achieve this, we encode the reference, Boolean, and integer attributes of classes in the start graph itself. An example of this can be seen in Figure 5.13, where we include all variables relevant to the PHILOSOPHER class. We call these constructs *object templates*.

To make use of object templates, we modify the semantics for object creation slightly. The rules action_New_From_Template and action_New_Local_From_Template handle object creation for attributes and local variables respectively. We attach the class name to **Action_New** nodes, denoting the type of object that we want to create. Then, the rule matches the template with the corresponding name, copies the template variables, and attaches them to the newly created object. The created object now contains attributes corresponding to its declared type. We can use arbitrary variable names as opposed to being restricted to generic ones as in CPM. In addition, objects now only have variables relevant to their types attached.

```
                                        ┌───────────────────────────┐
                                        │  RefOp_Retrieve_Ref       │
                                        │  var = "first_fork"       │
                                        └───────────────────────────┘
                                                    ↑
                                                  expr
                                        ┌───────────────────────────┐
                                        │  Param_Expr               │
                                        │  param = "Current"        │
                                        └───────────────────────────┘
                                                    ↑
                                                 target
   ┌───────────────────────────┐        ┌───────────────────────────┐
   │          Action           │        │  Action_Command           │
   │         command           │        │  method = "FORK.make"     │
   │  method = "FORK.make"     │        └───────────────────────────┘
   │  ref = "r_1"              │
   └───────────────────────────┘
```
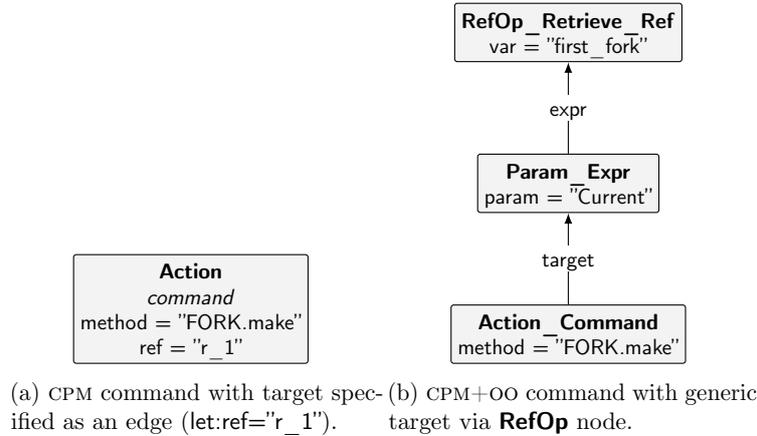
(a) CPM command with target spec-   (b) CPM+OO command with generic
ified as an edge (let:ref="r_1").   target via **RefOp** node.

Figure 5.14: Comparison of a command node in CPM and CPM+OO.

5.2.3 Generic Operators

In CPM, actions that require arguments, such as assignments or commands, are limited in that the arguments of these actions, including targets for queries and commands, can only be local data or reference variables, or simple operations like addition or negation. As a consequence, expressions that do not conform to this restriction have to be split up. For example, to represent the statement `account.withdraw (account.balance)`, we require two actions, the first one being a query for `account.balance` that is assigned to a (temporary) variable, and the second one being the command with this variable as an argument.

To enable more generic expressions, we change the representation of parameter nodes (see Figures 4.7 on page 26 and 5.2 on page 43 for the relevant type graphs of CPM and CPM+OO respectively). In CPM+OO, we can use the supertype **Param** instead of using specific types. We still have the **Param_Ref** and **Param_Data** types, as well as analogous **Param_Local_Ref** and **Param_Local_Data** types, which can be used to represent and fetch attributes and local reference and data values. In addition, and this is where the added flexibility comes from, we also provide a **Param_Expr** type, which has an expr edge to the **Super_Op** type. With this addition, we can now use arbitrary operations to fetch parameter data as opposed to only local and attribute values. Consequently, command actions and query operations do not specify the target via string any more. Instead, as seen in the type graphs in Figures 5.1 on page 41 and 5.6 on page 47, these nodes now have a target edge that points to a **Param_Expr** node which specifies the target. Once targets and parameters are evaluated, the behaviour is the same as if one would have used the **Param_Ref** and **Param_Data** types from CPM.

Figure 5.14 shows how a **RefOp** node can be used to specify the target, as opposed to a simple string with an attribute variable name (which is how targets are handled in CPM).

5.2.4 Lock Passing

Lock passing is necessary to avoid deadlock in certain situations. For example, when an object `a` holds the locks of the handlers of `b` and `c`, and creates a query request on `b` that in turn requires locking of `c`, then `b` can not proceed until `a` releases the lock on `c`, which in turn can not happen until `b` completes the request. No processor can make progress, and a deadlock has occurred.

To solve this problem, Morandi [13] defines a lock passing mechanism for feature calls. In his thesis, he defines feature calls as follows [13, p.19] (edited to reflect only relevant parts and for formatting):

A client p performs the following steps to *call a feature f* with target expression e_0 and argument expressions $e_1, \cdots, e_n$.

1. *Target evaluation.* Evaluate the target expression e_0 with supplier q.

2. *Argument passing.* Evaluate the argument expressions and bind them to the formal arguments.

3. *Lock passing.* Pass all locks to q if a controlled argument expression gets attached to an attached formal argument of reference type.

4. *Feature request.* Generate a feature request to apply f to the target.

   - If the feature call is non-separate, i.e. $p = q$, then ask q to process the feature request immediately using its call stack and wait for termination.

   - Otherwise, add the request to the end of q's request queue.

5. *Wait by necessity.* If f is a query, then wait for the result.

6. *Lock revocation.* If lock passing happened, then wait for the locks to come back.

To achieve this behaviour, we introduce a number of rules that interact with each other.

pass_locks This rule, shown in Figure 5.17, matches if at least one of the attached arguments is controlled, i.e., if the current processor (node $n0$) has a lock on some processor (node $n1$) that handles an object that is passed to the command. The target processor (node $n2$) receives all locks (lock edges) from the client processor. In addition, several edges are created for bookkeeping purposes. Edges labelled passed_lock point to processors that the client held a lock on and are required to know which locks to restore. The receiving processor has an edge restore_locks that points to the client, which allows giving back the locks to the correct processor later. Finally, an edge wait_for_restored_locks is added in order to make sure that the client does not continue execution before getting the locks back (which is ensured by a wait_for_restored_locks embargo edge in each action rule).

pass_locks_query Analogously to the previous rule, this one passes the locks for queries. While the previous one only passes locks if the arguments fulfill the requirements, this rule always passes the locks regardless of the arguments. This decision follows Clarification 5.4.2 in [13, p. 115].

Figure 5.15: Rule restore_locks_b.

restore_locks_... These two rules handle restoring rules for commands when lock passing has occurred. Figure 5.15 shows one of the rules, where the client (node $n8$) gets all the locks that have been passed earlier. To do this, edges restore_locks (which are created when creating the queue item with commands and queries that require lock passing) from the frame from the original request need to be present, which ensures that the locks get restored at the right point in the call stack even if the request triggers further feature calls.

cleanup_Restore_Locks_Query Similar as before, this rule (Figure 5.16) handles restoring locks of query requests.

Note that we do not support separate callbacks as described in the original semantics.

The above rules all have higher priority than action and query rules, which ensures that whenever the system is in a state where lock passing or revocation is required, it will apply the corresponding rule. It is therefore impossible for the system to miss passing the locks or continue without restoring the locks first. The latter is additionally enforced by adding embargo edges (wait_for_restored_locks in action and query nodes), which catches the case where locks are not properly restored due to no restore lock rule being applicable (this situation would indicate a problem with the CPM+OO model, not with the semantics or the inspected program).
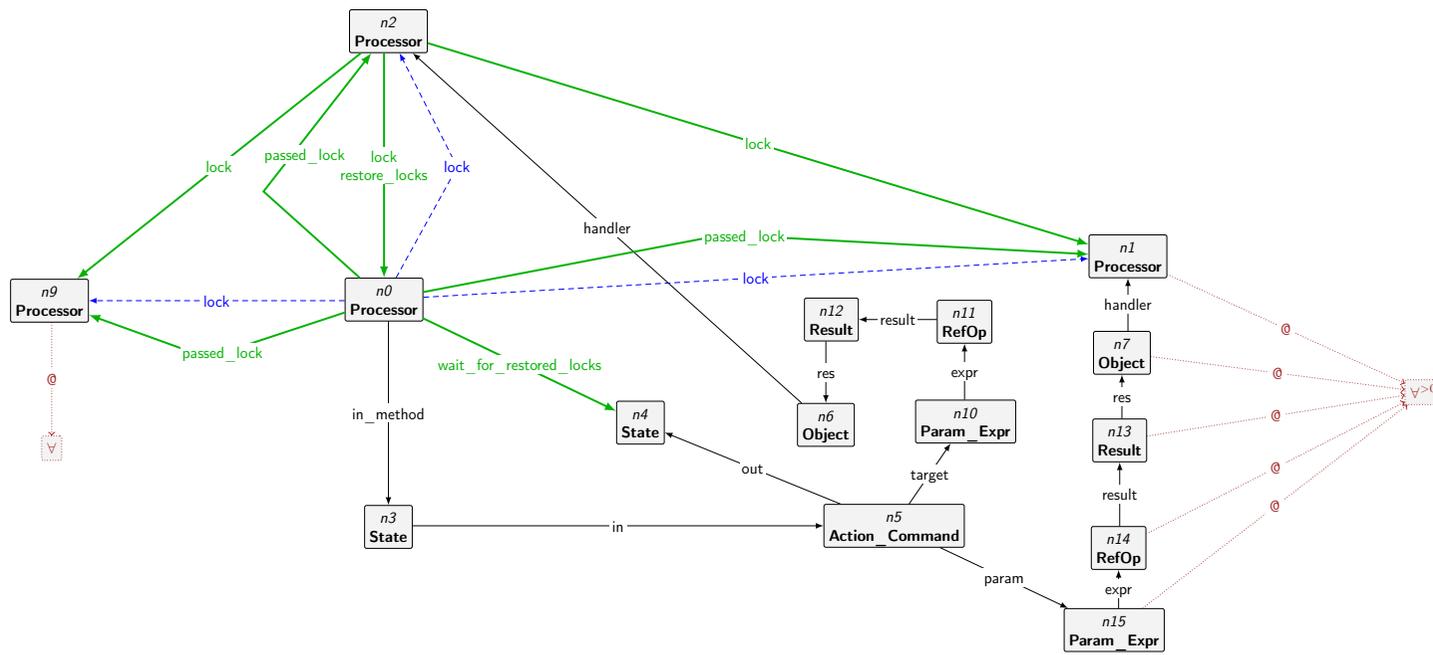
Figure 5.16: Rule cleanup_Restore_Locks_Query.

Figure 5.17: Rule pass_locks, that matches and is applied when at least one of the parameters is controlled.

5.2.5 Distinguishing Preconditions and Wait Conditions

In SCOOP, a statement in a `require` block involving separate arguments can either be a precondition or a wait condition, depending on the context from which it is called. If the processor executing the statement already held request queue locks to all involved processors of separate arguments before the call, then no other processor can enqueue requests to the queues of those handlers. As a result, the outcome of the statement can not change over time, and therefore it is a precondition. But in the case where the calling processor does not hold all locks, the result may change over time, and therefore the statement is a wait condition.

In CPM+OO, we distinguish between preconditions and wait conditions. The **Action_Test** nodes that denote the path that is to be followed if a precondition or wait condition evaluates to `False` are always marked with a *precondition_fail* flag. This does not necessarily mean, as described above, that the statement is in fact a precondition. Instead, the rule action_Test determines, based on the participating parameters, whether the statement is a precondition or a wait condition. In the first case, the rule creates an **ERROR** node, stating that a precondition has failed, and as a result, the system is in a final state. But if the statement turns out to be a wait condition, the rule handles it as such by following the action to the next state. Eventually, the processor returns the locks (which gives other processors the possibility to modify the state of the involved objects) before acquiring them again and evaluating the wait conditions again.

5.3 State-Space Optimisations

The state-space explosion problem is omnipresent in concurrent systems and therefore it is also present in CPM and CPM+OO. Obviously, this is an issue that one can not get rid of completely. Still, it can be of practical value to try and mitigate the problem as much as possible. We implement a number of optimisations that aim to reduce the state-space problem by avoiding unnecessary interleavings. Similar optimisations are already present in CPM, such as fine grained rule priorities for certain rules. Of course, we have to pay attention to the interleavings that are left out with these optimisations. In the following, we present additional measures taken to decrease the size of the state-space and argue why we are confident that they are not problematic with regards to the properties that can be verified using the CPM+OO model.

Quantifier Usage

In GROOVE, quantifiers can be used to create flexible rules, such as ones that match a type of node several times, or ones that express a logical "or", matching one part of the rule or another. While there are several situations where quantifiers are used in CPM, we extended this usage further in CPM+OO to help reduce the size of the state-space. The rule IntOp_RetrieveData (which is known as aexp_RetrieveData in CPM) is an example, shown in Figure 5.18, that uses a simple $\forall^>$ quantifier with several nodes attached, which means that the rule now creates **Result** nodes for each **Op_Retrieve_Data** attached to the action that is matched. In a situation with multiple **Op_Retrieve_Data**, CPM+OO creates all result nodes in one single rule application, whereas in CPM, the state-space
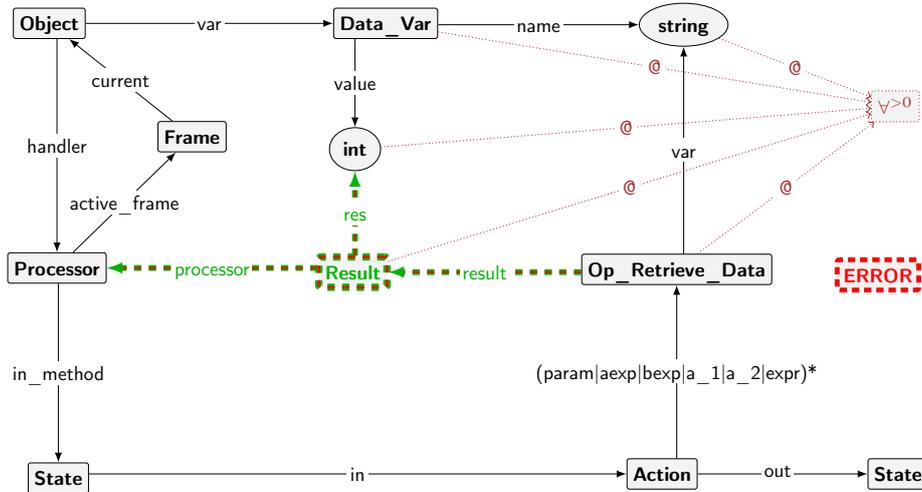
Figure 5.18: Rule IntOp_RetrieveData.

diverges at that point, exploring the different orders of creating these nodes, and converges once all result nodes have been created (since the resulting state is always the same, regardless of the order).

Scheduler Optimisation

Suppose we have a dining philosophers instance with only two philosophers (and thus with two forks). We are interested in the different interleavings that can occur with regards to locking, or more general, we are interested in the interleavings possible where philosophers and forks and their processors interact with each other by locking request queues. When simulating the instance with CPM or any version of CPM+OO, all the relevant interleavings are captured. Unfortunately, CPM and CPM+OO without scheduler optimisations also explore a large number of interleavings that we are not interested in, as they cover the same behaviour with respect to the outcome of the program. For example, one execution could execute the complete code of the first philosopher before the second one starts. In the next interleaving, the second philosopher might execute a (local) identifier assignment, then the first one executes everything, before the second one executes the remaining part. With respect to the interaction of processors and objects, these two interleavings are equal.

More generally, we can split actions and queries in two groups: non-separate and separate. In the first case, everything is handled on the current processor, no other processors are involved. In the latter case, different processors may be included and therefore, we want to explore all possible executions (since the outcome of executing SCOOP programs is determined by the order in which requests are enqueued). The idea behind this optimisation is that as long as a processor is executing locally (e.g. a philosopher initializing by performing integer assignments for the identifier and number of times to eat attributes) without having an impact on any processor's request queue, we can advance it as far as possible. Once a processor is at a point where a non-separate action or

query is about to be executed, it waits until all other processors have reached a similar position (or have finished executing and are idle). All processors that are not idle are potentially about to interact with other processors. At this point, it is important to explore all interleavings, as different orders in locking and enqueuing requests may result in different situations (e.g. in the dining philosophers example, when the first philosopher has a fork and is about to pick up the second, but the second philosopher is about to pick up the same fork, we want to explore both situations where one or the other philosopher "wins").

Implementation We implement this idea by introducing an *execution token* and organising the processors in a linked list. In a CPM+OO state, at most one processor has a *token* flag, which denotes that it is allowed to perform non-separate steps. To achieve this, we give the non-separate rules a higher priority and add the token as a requirement to match. These rules (e.g. rule action_Command_non-separate) can only be executed by the processor that has the token. Once a processor can not perform more non-separate actions, it passes the token to the next processor. This is repeated, until no processor can make non-separate progress any more. Once this is the case, the separate rules (such as action_Command_separate) can be applied. These do not require the token, which means multiple rules may be applicable in a given state, and since these rules all have the same priority, all interleavings are explored by the system.

In the following, note that non-separate rules have priority 4 and separate rules have priority 1. The rules pass_token, pass_token_first, and reset_token are relevant for this mechanism. These rules with priorities 3, 2, and 0 respectively handle the movement of the token flag along processors and are shown in Figure 5.19. The token is cycled until there is one full cycle where no processor has made progress. To achieve this, we use a node of type **Action_Executed_Indicator**. Such a node is created whenever a non-separate action is performed, e.g. by the rule action_AssignRef. When the token is on the last processor, it gets moved to the first one (thus restarting the cycle) only if there is an indicator node. When the token has run through the list without an action having been performed, the rule is not applicable anymore, which enables rules with lower priority, in particular the separate rules. At this point, all different interleavings between separate rules are explored as intended. Similar to the **Action_Executed_Indicator**, separate actions create a **Reset_Token** node that indicates that a separate step has been performed, which means that some processor may possibly continue with non-separate steps. If such a node exists, the reset_token rule can be applied which removes the node and puts the token back on the first processor, restarting the cycle to perform non-separate steps. In case there is no such **Reset_Token** node, the rule cleanup_token is applied that removes the token from the processor holding it, ensuring that there are not several final configurations with the only difference being that the token is on a different processor.

As we will see later in Chapter 7, this optimisation results in a huge improvement in the size of the state-space and allows us to verify programs where the state-spaces have been too big before.

We argue that this mechanism does not leave out interleavings of interest, as all the possible sequences in which requests get added to queues are preserved. By forcing a certain order for local computations, where the order of execution
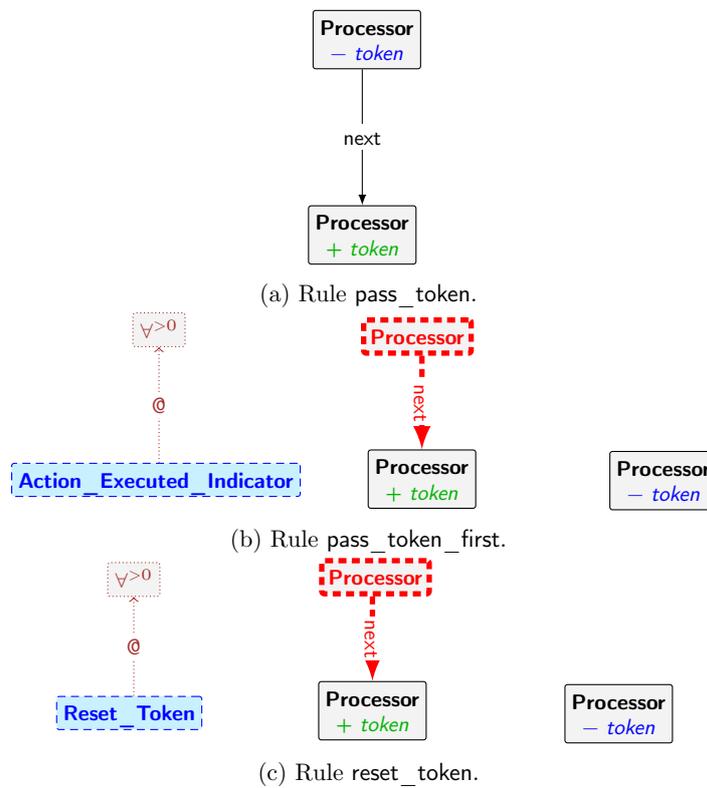
(a) Rule pass_token.

(b) Rule pass_token_first.

(c) Rule reset_token.

Figure 5.19: Rules handling token movement.

has no influence on the outcome, we can avoid exploring a large number of states.

5.4 Rules

Here, we provide a complete overview of all rules and their priorities in the CPM+OO system. Together with the type graph we have presented, this makes up the complete GTS of CPM+OO. While we discuss certain rules in detail, we do not show graphs for every rule, but instead refer the interested reader to the supplementary material repository [21]. In addition, we also present the priorities of the rules as we have done for CPM.

We divide the rules in CPM+OO into the following categories and discuss them in the subsequent sections.

- Control Flow

- System State

- Queries and Other Operations

- Optimisations

- Errors

- Configuration

5.4.1 Control Flow

Control flow rules handle movement along feature graphs, in particular moving from a state via an action node to the next state. Table 5.1 summarizes these rules and their priorities. Most rules have direct counterparts in CPM, although new variants have been introduced to handle additional features like non-separate calls. What follows is a short discussion of the rules.

action_Assign_Data This rule handles an integer or Boolean assignment operation where the target is an attribute of the current object.

action_Assign_Local_Data Similarly, this rule handles integer and Boolean assignments to variables declared in the `local` block. The rule accesses local data stored on the frame instead data stored on the object, since local declarations are only valid within the context of the current call and therefore of the current call stack frame.

action_AssignRef Analogously to the previous two, this rule handles reference assignments. While CPM uses a number of rules for reference assignments (action_AssignRef_Ref_... rules), the CPM+OO rule is not injective, and we make additional use of quantifiers to express alternatives, which makes it possible to cover all cases with a single rule.

action_Assign_Local_Ref Analogously to action_Assign_Local_Data, this rule handles the case for reference assignments to local variables.

action_AssignResult_... For each data type, CPM+OO has a rule for assignment to the special `Result` value, representing return values in queries. Figure 5.20 shows the rule for reference values.

action_Command_non-separate

action_Command_separate

action_Command_separate_restore_locks The non-separate command rule
creates a new stack frame, sets it up with parameters, and puts it on top
of the frame stack of the current processor. In addition, the rule points
the current processor to the designated feature. This represents a local
call that is executed immediately. The separate cases on the other hand
create a feature request and attach the created frame to it. The request
is attached to the target processor and will then get processed by queue
management rules. The calling processor can proceed since the separate
rules only match if the target processor differs from the calling processor,
which means that the command is asynchronous.

action_CreateRoot Since cpm+oo allows specifying the root class and pro-
cedure, we need a rule that creates the initial object, which is what
action_CreateRoot does. The configuration node with root class name and
procedure name is deleted in this rule, ensuring that only one root object
is created. The object is created according to the class template, similar
to what the rule action_New_From_Template does.

action_New_From_Template

action_New_Local_From_Template To instantiate attributes and local vari-
ables, these rules match **Action_New** nodes and their context. The created
object is then attached to the specified variable, where the first rule handles
attributes and the second one local variables.

action_Lock The lock action rule takes, as opposed to the lock rules in cpm, a
variable number of references. The rule can only be applied if all handlers
of the specified objects are not locked. Applying this rule implies that all
locks are obtained atomically. Figure 5.21 shows the rule graph.

action_Test A form of branching is provided with the test action, which works
analogously to the cpm test action.

action_Noop This rule simply skips an **Action_Noop** node and, as the name
indicates, performs no real operation.

action_TestPostcondition Like the cpm rule of the same name, this one ad-
vances a processor in a final state to the first state of the postcondition, if
the graph is configured to check postconditions.

action_Unlock_Creator

action_Unlock_Creator_non-separate Since created objects are locked by
their creators, they need to have an unlock action as their last action
in creation procedures, which removes the lock, allowing the creator to
continue execution (since the creator immediately, by convention, has to
follow with a pair of lock and unlock actions for the same object that was
just created). These rules handle the separate and non-separate case.

action_Unlock_Expr If a feature obtains locks at the start, there are corresponding unlock actions that release them at the end of the feature. This rule handles the unlock actions. Not only does it release held locks, but in case the lock is not held (which can happen if several passed separate arguments have the same handler) the action becomes an empty operation.

Aside from these rules, the rules involved in lock passing belong to this group. They are not repeated here, instead we refer to Section 5.2.4, where they are described in detail.
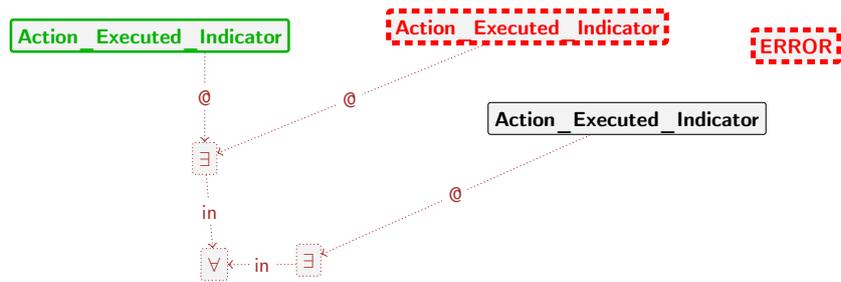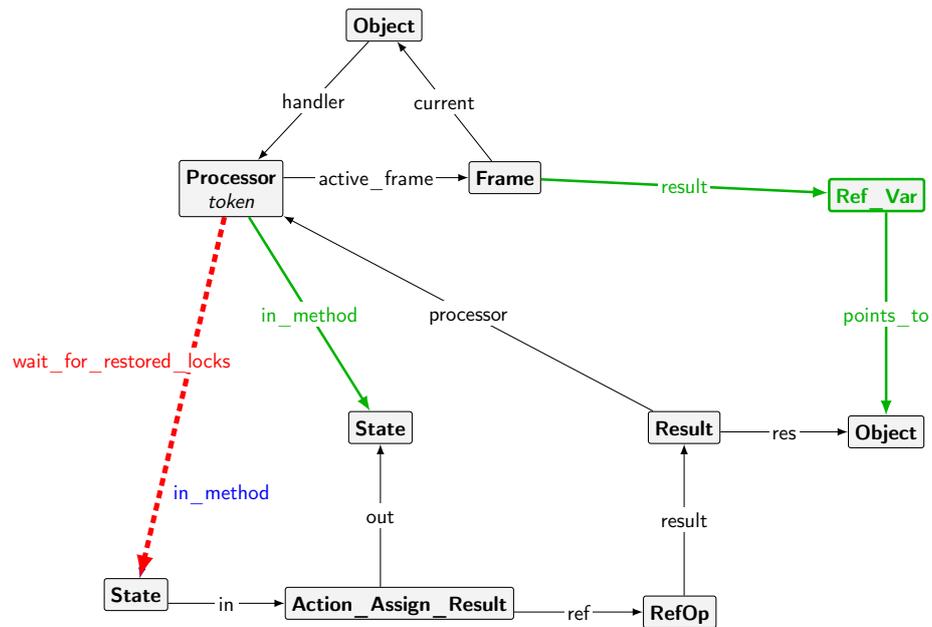
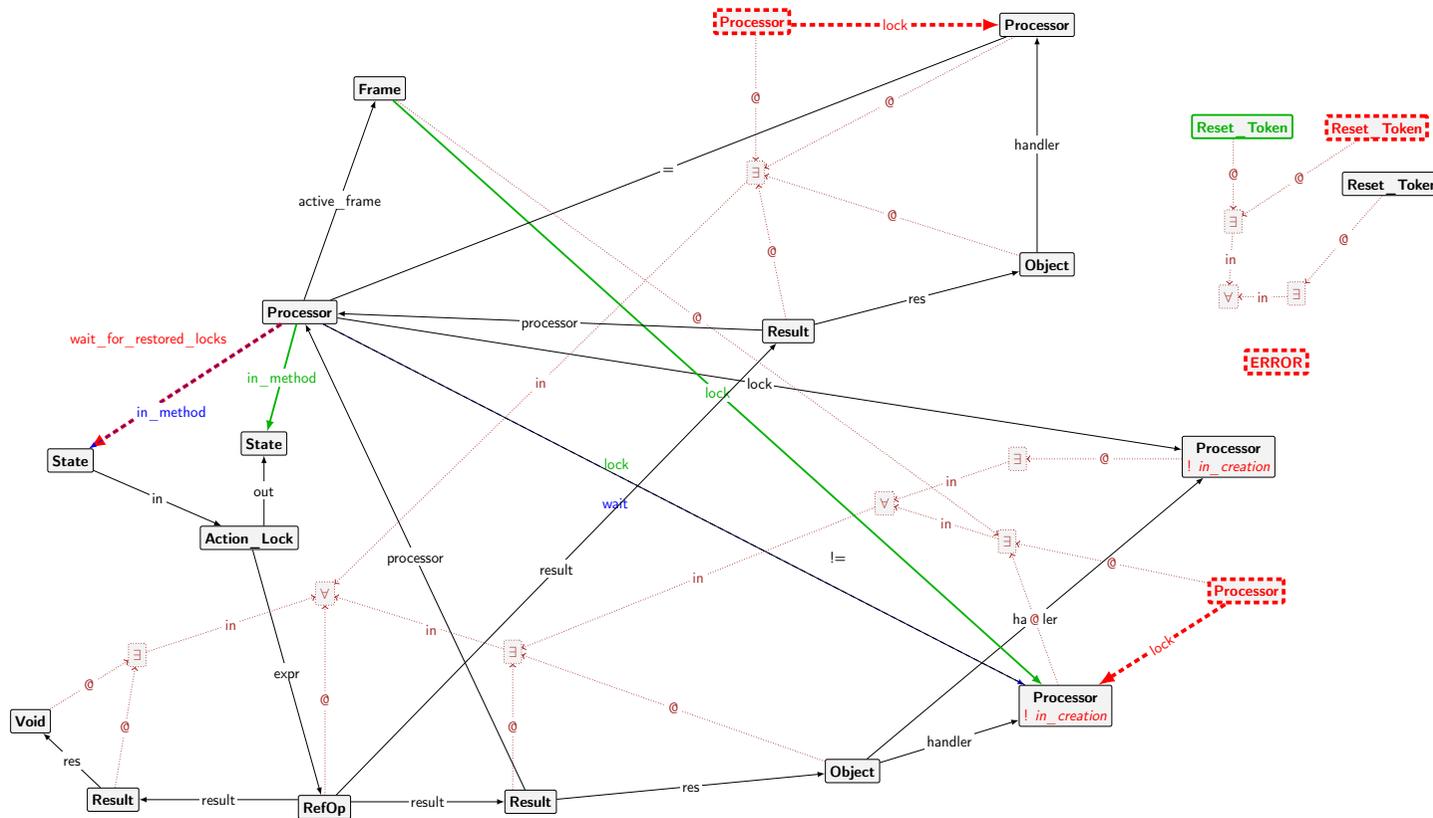Figure 5.20: Rule action_AssignResult_Ref.

Figure 5.21: Rule action_Lock.

Rule	Priority
restore_locks_a	71
restore_locks_b	70
action_TestPostcondition	20
prepare_lock_wait	19
pass_locks_query_new	18
pass_locks_	17
action_Assign_Data	6
action_Assign_Local_Data	
action_Assign_Local_Ref	
action_Assign_Ref	
action_AssignResult_Bool	
action_AssignResult_Int	
action_AssignResult_Ref	
action_Command_non-separate	
action_CreateRoot	
action_New_From_Template	
action_New_Local_From_Template	
action_Noop	
action_Test	
action_Unlock_Expr	
action_Command_separate	1
action_Command_separate_restore_locks	
action_Lock	
action_Unlock_Creator	
action_Unlock_Creator_non-separate	

Table 5.1: Control flow rules.

5.4.2 System State

The rules in the system state group, listed in Table 5.2, are concerned with queue management and graph maintenance. The former includes rules that insert queue items into the request queue and remove them when processing an item. The latter deal with various graph states with leftover nodes, e.g. when a processor has reached a final state and results need to be discarded.

cleanup_exp_DiscardResults_BoolOp

cleanup_exp_DiscardResults_Op

cleanup_exp_DiscardResults_RefOp

cleanup_exp_DiscardResults_RefOp_Void After evaluating an operation (in the form of **Super_Op** nodes), it has a **Result** node attached which is then used by the action to process. Once a processor moves past the action node, these rules are are applied to remove the **Result** nodes since they are not used anymore.

cleanup_remove_Void In certain situations, it can happen that **Void** nodes are left without being connected to any part of the graph. These are removed by this rule in order to avoid creating several states in the LTS that only differ in the amount of unconnected **Void** nodes.

cleanup_Frame_Remove_controls Frames can have controls edges to processors which have been controlled prior to the call. This is required to

determine whether a statement in the `require` block is a pre- or wait condition.

cleanup_FinalState_... A number of cleanup rules are applied once a processor reaches the final state of the procedure it is executing. They perform a range of tasks, including removing the current frame or setting the result value such that the calling processor has access to it. The following rule exist in this set.

- cleanup_FinalState
- cleanup_FinalState_with_Return_Value
- cleanup_FinalState_BoolQuery
- cleanup_FinalState_BoolQuery_with_next_frame
- cleanup_FinalState_Command_Empty_Call_Stack
- cleanup_FinalState_Command
- cleanup_FinalState_IntQuery
- cleanup_FinalState_IntQuery_with_next_frame
- cleanup_FinalState_Local_Data_Objects
- cleanup_FinalState_Local_Ref_Objects
- cleanup_FinalState_Param_Data_Objects
- cleanup_FinalState_Param_Ref_Objects
- cleanup_FinalState_RefQuery
- cleanup_FinalState_RefQuery_with_next_frame

queue_Insert_EmptyBusy

queue_Insert_NotEmpty When a client creates a request queue item, it does not actually insert the item directly into the request queue. Instead, rules like action_Command_separate simply let the **Queue_Item** point to the target processor via insert_into edge. The actual insertion into the queue, depending on whether it is currently empty or not, is performed with the queue_Insert_EmptyBusy and queue_Insert_NotEmpty rules respectively.

queue_Remove_SingleQueued

queue_Remove_MultipleQueued Once the request queue has items, these rules are used to remove a queue item from the request queue and instruct the processor to start execution at the designated procedure. The first one handles the case where exactly one item is on the queue, the second one cases with more than one item on the queue.

prepare_lock_wait Before a lock action is performed, a wait edge is inserted from the processor executing the action to the processors it intends to lock. These edges are in particular useful for detecting deadlock with the rule error_deadlock. These edges are deleted once the target processors are locked.

Rule	Priority
cleanup_Remove_Void	700
queue_Insert_EmptyBusy	600
queue_Insert_NotEmpty	590
cleanup_exp_DiscardResults_RefOp_Void	461
cleanup_exp_DiscardResults_RefOp	460
cleanup_exp_DiscardResults_Op	450
cleanup_exp_DiscardResults_BoolOp	440
queue_Remove_MultipleQueued	160
queue_Remove_SingleQueued	159
cleanup_Restore_Locks_Query	69
cleanup_Frame_Remove_controls	66
cleanup_FinalState_IntQuery	65
cleanup_FinalState_IntQuery_with_next_frame	64
cleanup_FinalState_RefQuery_with_next_frame	63
cleanup_FinalState_RefQuery	62
cleanup_FinalState_BoolQuery_with_next_frame	61
cleanup_FinalState_Objects_with_Return_Value	60
cleanup_FinalState_BoolQuery	59
cleanup_FinalState_Local_Data	58
cleanup_FinalState_Command	57
cleanup_FinalState_Command_Empty_Call_Stack	56
cleanup_FinalState	55
cleanup_FinalState_Param_Data	53
cleanup_FinalState_Param_Ref	52
cleanup_FinalState_Local_Ref	51
remove_wait_and_lock	16

Table 5.2: System state rules.

remove_wait_and_lock When a processor is in a state before a lock action, it first creates a wait edge that points to the processor it intends to lock. The rule action_Lock can only be applied if for all target processors, either the lock is already held, or a wait edge exists. In the former case, the graph is not modified any further. This rule handles the situation where a processor has both a wait and a lock edge, in which case the wait edge simply gets deleted.

5.4.3   Queries and Other Operations

As in CPM, we group rules related to queries and operations on integers, Booleans, and references together. Table 5.3 lists all rules in this group. A description of rules and rule families follows.

BoolOp_Query_ ... The Boolean query rules handle separate and non-separate queries, where a new frame is created. In the separate case, a request queue item is created and attached to the target processor, whereas in the non-separate case, the frame is put on the current processor's frame stack and the processor is instructed to start executing the query.

BoolOp_RetrieveData Similar to other RetrieveData rules in both CPM and CPM+OO, this rule fetches attributes of the current object of Boolean types.

BoolOp_ ... Other Boolean operations include constants, conjunction, disjunction, equality, and others. These rules, with their arguments evaluated, perform the corresponding operation and attach the result to the matched **BoolOp** node.

IntOp_ ... Similar to the rules handling Boolean operations, these rules handle various integer operations, such as simple addition.

RefOp_ ... Analogously, a number of rules handle fetching references. This includes getting attributes or local references, but also creating query requests.

getlocal_ ... The getlocal_Data and getlocal_Ref rules prepare instances for local variables, which are attached to the created frame later when a command or query is called.

getparam_ ... Once the values of **Param** nodes have been evaluated, these rules are applied to create instances in the form of **Param_Data** and **Param_Ref** nodes which are then passed to the called query or command.

5.4.4   Optimisations

Optimisation rules are those involved in handling the execution token discussed earlier, and are listed along with their priorities in Table 5.4. A thorough discussion of the involved types and rules is given in Section 5.3.

5.4.5   Errors

In this group of rules, we collect error conditions. This includes properties such as presence of a deadlock or a void call, which we are interested in when verifying programs. In addition, we also have rules that aid us during development and serve as "sanity checks". For example, the rule debug_multiple_handlers matches, if an object has more than one handler. Since this situation is not possible according to the SCOOP specification, a match of this rule means that there is an error in our model. Matching such "bad states" was used extensively during development to catch bugs, but the corresponding rules have been removed from the final GTS.

   The priorities of the current error rules are listed in Table 5.5 on page 74, a short description of them follows.

error_deadlock A large part of the motivation behind this work is detecting, amongst other properties, deadlocks in SCOOP programs. This rule, shown in Figure 5.22, detects deadlocks by matching, if a processor $n1$ has a lock on some processor $n4$, but is also waiting on a processor $n2$, which in turn is locked by some other processor (not shown, but expressed using the regular expression edge -lock.wait)+) which again is waiting on $n4$. An example configuration where this rule matches is shown in Section 7.2.1.
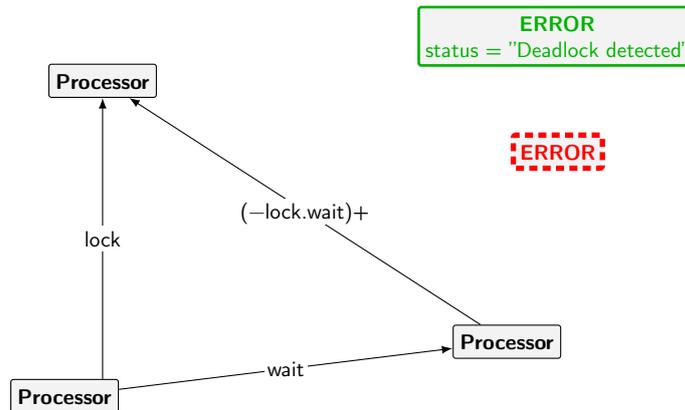
error_deadlock_2 and error_deadlock_3 Deadlock situations can not only occur in the above case where no processor is able to acquire locks and make progress. For example, when two processors execute the same feature

Rule	Priority
getparam_Expr_Op	432
getparam_Expr_BoolOp	431
getparam_Expr_RefOp	429
getparam_Local_Ref	413
getparam_Local_Data	412
getparam_Data	411
getlocal_Data	410
getlocal_Ref	409
IntOp_constant	400
BoolOp_constant	390
RefOp_RetrieveRef_Local	386
RefOp_RetrieveParam	385
RefOp_RetrieveRef_Void	384
RefOp_RetrieveRef	383
IntOp_RetrieveParam	382
IntOp_RetrieveLocalData	381
IntOp_RetrieveData	380
IntOp_RetrieveData_with_Target	379
IntOp_Multiply	351
IntOp_Add	350
IntOp_Subtract	340
BoolOp_RetrieveData	334
BoolOp_And	333
BoolOp_GreaterEquals	332
BoolOp_Equals	331
BoolOp_GreaterThan	330
BoolOp_Equals_Ref_False	329
BoolOp_Equals_Ref_True	329
BoolOp_Equals_Ref_Void	329
BoolOp_LessEquals	321
BoolOp_LessThan	320
BoolOp_Not	310
BoolOp_Query_debug_non-separate	0
BoolOp_Query_separate	
BoolOp_Query_separate_restore_locks	
IntOp_Query	
IntOp_Query_separate	
IntOp_Query_separate_restore_locks	
RefOp_Query_non-separate	
RefOp_Query_separate	
RefOp_Query_separate_restore_locks	

Table 5.3: Query and operation rules.

Rule	Priority
pass_token	3
pass_token_first	2
cleanup_token	0
reset_token	0

Table 5.4: State-space optimisation rules.

Figure 5.22: Rule error_deadlock.

Rule	Priority
debug_Multiple_Handlers	1000
error_Command_Void_Target	
error_Deadlock	
error_Deadlock_2	
error_Deadlock_3	
error_Postcondition_Fail	
error_Query_Void_Target	

Table 5.5: Error rules.

which contains a wait condition that requires the other processor to finish this particular feature, then both processors can lock the request queue of the other one. They then both wait for the other one to handle the query request generated in the wait condition and therefore none of them makes progress. The rules error_deadlock_2 and error_deadlock_3 handle such situations for two or more processors respectively.

error_PostconditionFail If a postcondition is evaluated to `False`, this rule puts the processor in a special state of type **State_Postcondition_Fail** and creates an **ERROR** node with attached information about where the postcondition has failed.

error_Command_Void_Target

error_Query_Void_Target If a target of a command or query has been evaluated to a void reference, then the call is invalid, which is detected and reported with these two rules.

5.4.6 Configuration

Currently, there is only one rule in this category, the rule config_CheckPostcondition, which is applied if one specifies that postcondition should be checked. It has a high priority and advances a processor from a final state to the start of

the postcondition, ensuring that no cleanup rules are applied before the postconditions have been checked. The processor will evaluate the postconditions and if everything evaluates to true, end up in another final state where the normal cleanup rules can be applied. If no such configuration node exists, this rule can not be applied and the cleanup rules take place, ignoring possible postcondition related parts of the graph.

5.5 Testing

The CPM+OO model has been developed in an iterative fashion by adding features described in this chapter to the CPM model one by one. Changing the GTS is error-prone. It is all too easy to alter the behaviour such that it does not reflect the intended one any more by adding rules that contain bugs, changing priorities that result in certain rules being applied in a state where we do not want the rule to be applicable, or altering the type graph and rendering existing rules useless. To ensure that the model stays true to the intended behaviour, we use a number of start graphs representing test programs and specify the expected output. For example, along with evolving the model, we also evolve the examples of the dining philosophers with both the correct and the deadlock implementation. Our testing utilities then explore the state-spaces of these examples and match it against the expected behaviour which checks properties like state-space size, the number of final configurations, and whether **ERROR** nodes are present in final configurations.

Once we finalised the type graph for the current CPM+OO model, we used this testing approach in combination with our translation tool, described in Chapter 6. This allows us to write SCOOP programs and specify the expected output of our state-space exploration tool. The testing utility then first translates the source code to a CPM+OO start graph, and then explores the state-space and checks whether the actual output matches the expected output.

5.6 Future Work

With CPM+OO at its current state, we are able to simulate a number of SCOOP features directly in the model, as opposed to simulating them using more basic CPM constructs. We added rules and types to CPM that make the model more expressive and allow start graphs that closely resemble the corresponding SCOOP source code.

To support more SCOOP features, one possible way is to extend the CPM+OO model to directly support those features. This has the advantage that programs that make use of those features can be represented directly in a compact and readable fashion. Another approach is simulating these features using existing CPM+OO functionality. In our automatic translation tool, it would require additional work to express features not directly supported by CPM+OO, resembling the work of traditional compilers.

We have several strategies in mind on how to implement certain missing features. Since they often not only involve considerations regarding the CPM+OO model, but also the translation tool discussed in the next chapter, we postpone a more thorough discussion of future work until Section 6.6.

Chapter 6

Translation

With CPM+OO, we introduced object-oriented features of SCOOP to the CPM model. Thanks to that effort, more SCOOP programs can now be represented and simulated using the model. Since both CPM and the extensions that we introduce in CPM+OO are closely modelled after SCOOP, mapping source code to start graphs becomes a less tedious task. In this chapter, we discuss the automatic translation tool that translates a subset of SCOOP to CPM+OO start graphs.

6.1   Overview

Translating a SCOOP program to CPM+OO consists of a number of steps, as depicted in Figure 6.1 where the tool progresses from top to bottom. In the first step, SCOOP source files are parsed and syntax trees are generated. Using these syntax trees, an internal representation of the program is created in two steps: First the syntax trees are walked to gather typing information of features and variables. Then, in a second pass through the syntax tree, we use typing information to create a structure that closely relates to the CPM+OO type graph. In the final two steps, the intermediate representation is transformed to a simple graph representation, which also contains layout information. Finally, this graph can be traversed and rendered as an XML file that can be used in the CPM+OO transformation system.

The tool is implemented in Java and uses a number of libraries, namely the following.

ANTLR 4.4 *ANother Tool for Language Recognition* (ANTLR) is a parser generator that, given a grammar in *Extended Backus-Naur Form* (EBNF), generates a lexer and parser in Java. The created classes offer a large amount of flexibility and implement the visitor pattern, providing a natural way to traverse the parse tree. While this is possible with other parser generators as well, the modern and clean nature of the generated classes have convinced us to use ANTLR for this project.

JUnit The JUnit framework is used to automatically test various aspects of the implementation.

Apache Commons The commons libraries offer a wide variety of reusable

software components. This project makes use of the mathematics features, in particular for case studies and evaluation purposes.

GROOVE Not only does GROOVE provide graphical and command-line interfaces, but it can also be used as a library in custom software. We use the library to perform exploration and verification from within our toolchain. This enables us to create more specific output tailored to CPM+OO as opposed to the generic GTS output provided by the command-line interface of GROOVE.

What follows are more detailed technical descriptions of the individual steps.

6.2 Translating Programs

With the help of ANTLR, the first step of parsing consists of writing a grammar in the ANTLR grammar format. We did not write this from scratch; instead we adapted the grammar found in EVE [22] for this usage. During this process, we modified the grammar to conform to the ANTLR file format. Given this grammar, ANTLR is able to generate a lexer and a parser which can be used in our tool.

It is important to note that at this stage, we consider all SCOOP programs. This means that we are able to parse programs with more advanced features, such as inheritance and generics. The decision whether a program is translatable or whether it contains unsupported features is performed when inspecting the syntax tree (steps 2a and 2b in Figure 6.1).

To perform step 2a, the tool uses a class that implements the SCOOP syntax tree visitor. It keeps track of the class currently inspected and stores the following type information for each class:

- Declared routines and their parameter types and return types (if any).

- Declared attributes and their types.

- A list of creation procedures.

It is necessary to record this typing information in advance, as we need to know the types of symbols in the next step. In a single-pass approach, it is possible to encounter symbols from classes that have not yet been analysed, therefore making it impossible to know whether it is an integer, Boolean, or reference symbol.

After having gathered the types of declarations, we pass through the parse tree once again. This time, we create a number of `CPMOGraph` objects, one for each parsed class. The `CPMOGraph` class and its subclasses are closely related to the CPM+OO type graph. In fact, most types in CPM+OO have a direct representation as a subclass of `CPMOGraph`. For example, we use a class `BoolConstant` that inherits from `BoolOp`, which in turn inherits from the class `Op`. Similarly, in CPM+OO the `BoolOp_Constant` type is a subtype of `BoolOp`, which in turn is a subtype of `Super_Op`. This direct correlation is useful in a variety of ways. In particular, translating a `CPMOGraph` to a CPM+OO start graph is straightforward, as we can simply go through the structure and create a CPM+OO graph node (in a generic graph representation) for each encountered `CPMOGraph` object. This also means that the "compiler effort", i.e. relating source code statements (in the form of parts of the syntax tree) to CPM+OO nodes is concentrated in one

*.e files

Parser

1   generated by ANTLR parser gen-
    erator

Syntax Tree

2a   collect typing information

Type Information

2b

2b   create CPM+OO representation
     using gathered type information
     to distinguish integers, Booleans,
     and references

CPM+OO Objects

3   create simple graph representa-
    tion with layouting information

Graph

4   render as GXL for usage in
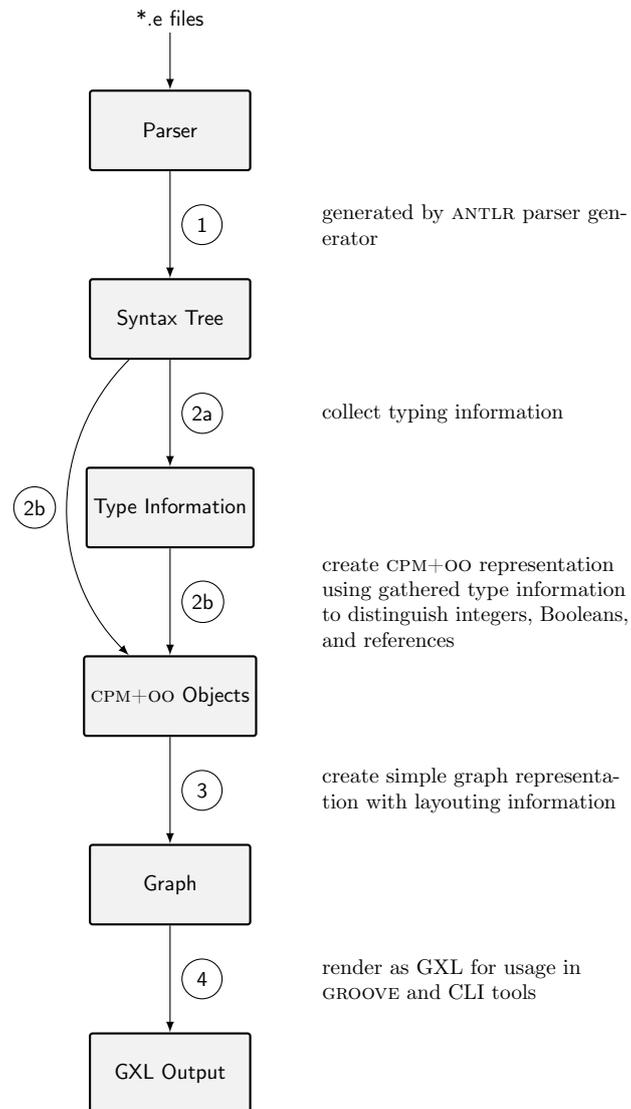   GROOVE and CLI tools

GXL Output

Figure 6.1: Overview of the steps included in translating a set of Eiffel classes to
a CPM+OO graph rendered as a GXL file.

step, namely the visitor class that implements step 2b. This second visitor also handles situations of input programs that use features currently not supported. If our tool encounters such a feature in step 2b, it either ignores it (in cases where the feature does not influence the program execution, e.g. a `note` block at the top of a class) or aborts the translation and prints the part of the source code that resulted in the tool to fail. This way, we have a single point in the tool where the decision is made whether a program is supported, and only one point at which translation can fail due to the nature of the input program.

6.3 Supported SCOOP Features

For correct translation and simulation in CPM+OO, we require a complete SCOOP program to be passed as a set of input files. In particular, all referenced classes must be part of the input. A number of other restrictions on the input programs apply for the tools to function correctly. In this section, we give an overview of the supported features of SCOOP and discuss the parts that are missing.

The translation tool focuses on the basic features of SCOOP. The goal is not to support the complete SCOOP language, but enough of the language to allow writing expressive programs in an object-oriented manner. The following features are currently supported.

Classes and Objects CPM+OO supports objects and classes natively. The translation creates object templates for each input class consisting of the class name and the names and types of its attributes.

Feature declarations Both routines (with the `do` keyword) and attributes are translated. While attributes are part of the class template, we also create a getter routine (consisting of simply assigning the attribute to `Result`) for each attribute. This makes it possible to create **Queue_Items** that call the getter functions instead of accessing the data from other processors directly (which would cause CPM+OO to misbehave, as the requests would not be served in FIFO order anymore).

Routine declartions In routines, we support common constructs such as formal arguments, preconditions, wait conditions, and postconditions.

Local declarations Local variables of reference, integer, and Boolean type are supported. Integer and Boolean types are special native types in CPM+OO and behave like expanded types. Generic support for expanded types is currently not available.

Instructions A number of instructions are supported, namely:

- Creation calls with the `create` keyword and an explicit creation procedure.
- Local calls (with target `Current`).
- Qualified calls.
- Assignment instruction.
- `if`-`then`-`else` conditional.
- Loop instruction.

- Integer and Boolean literals.

Expressions We support arbitrarily complex expressions, which are translated to a single **Op** node in the output graph. As a consequence of leaving out a number of features such as agents, expressions related to those features are not supported (e.g. agents inside expressions).

Both CPM+OO and the translation tool currently lack support for a number of features. Most prominently, we do not support inheritance. With this, a number of related SCOOP features are not supported either, for example partial classes, the redefinition of features, arrays, generics, agents, and others. In addition, we currently leave out a number of other language features, such as class invariants, old values in postconditions, and others. In Section 6.6, we give an overview of the most important features currently missing and present possible implementation strategies as future work.

6.4 Output

Once the intermediate representation of the input files is generated (in the form of `CPMOGraph` objects), the remaining task is outputting the representation as a GXL file. To achieve this, we use the visitor pattern once again: The interface `CPMOGraphVisitor` allows implementing classes that pass through the CPM+OO structure. This is used to create a simple graph representation using the `output.graph.Graph` class and its subclasses. These classes implement a straightforward graph representation with nodes and directed edges. In addition, nodes can also store position values. This allows us to create start graphs that are "human readable" when rendered in GROOVE. In particular, we organise routine subgraphs by aligning states and actions from left to right, while attaching additional nodes, like parameters and target operations, above them.

Using two separate steps to output a `CPMOGraph` structure to XML may seem unnecessary, as we could as well just have generated the GXL file directly. But using a separate simple graph representation has the advantage that we can separate the tasks of creating graphs with layout information and rendering them in some format (in our case GXL). This leaves more flexibility when extending the program, for example when we want to render the graph in another output format we can traverse a simple graph structure with edges and positioned nodes. When extending the CPM+OO model, we simply have to adjust the part that generates a graph from `CPMOGraph` objects, but do not have to adjust anything related to the GXL output. This leaves us with a well structured design that cleanly separates concerns and can easily be extended at various stages.

6.5 Testing

As briefly mentioned in Section 5.5, we test our translation tool in conjunction with the model by providing SCOOP programs, translating them, and exploring their state-spaces. The output is matched against the expected output using the JUnit framework. With this approach, the start graph is implicitly tested against the type graph presented in Section 5.1. By assuming that the model behaves correctly at this point, we can test the translation tool by simulating

the generated start graph and analysing the output. In case the output does not match, we most likely have an error in the translation tool.

We do realise that this is hardly "unit testing" in the traditional sense, instead we test the toolchain as a whole. While this may be suboptimal in general, we are confident that it is sufficient for the size of this project and due to the fact that this is a prototype implementation. In addition, we develop only a single part of the toolchain at a time, i.e. we either change the translation tool or the CPM+OO model, which allows us to check the influence of the changes on the final output.

To make sure that we catch the expected behaviours when translating and modelling, we use a wide range of test input programs and specify the expected behaviour. This includes small programs that focus on certain features, e.g. ones that use a wide range of available query types, as well as larger example programs that resemble real programs, such as the ones used in the case studies in Chapter 7.

6.6   Future Work

In Section 5.6, we briefly discussed features missing from the CPM+OO model, and in Section 6.3 we named some SCOOP features that are not handled in the translation tool. In this section, we propose ideas to how certain features could be implemented in the future. Since this not necessarily only affects the translation to the CPM+OO model, but may require changes in the model itself, we discuss possible changes to the CPM+OO model as well.

In general, supporting additional features can be tackled by either extending the compiler to translate to the current CPM+OO model, which means that the feature is simulated using more primitive CPM+OO constructs, or by extending CPM+OO itself by adding direct support of these features. The advantage of the latter is that program representations become easier to read and understand, and a more direct translation can be made from source code to start graph. While this is a desirable outcome, it also requires careful reasoning about the model changes, something one can avoid if only the compiler is extended.

6.6.1   Inheritance

The most important feature towards supporting more complex SCOOP programs is inheritance. The main difficulty in supporting inheritance is the complexity and feature-richness of the semantics related to inheritance. SCOOP offers a wide range of mechanisms, such as multiple inheritance, redefining, undefining, and renaming of features, partial classes, and others. As a result, adding these features to either the translation tool or CPM+OO requires careful analysis of the underlying semantics. Identifying and isolating individual parts of the inheritance mechanisms and modelling them one by one (where possible) seems to be the right approach to tackle this task, which allows us to be confident in the resulting model.

To implement simple inheritance (i.e. using the `inherit` keyword), one strategy would be to "unfold" the inheritance structure during translation. This means that for a class `FOO` that inherits feature `baz` from class `BAR`, we simply create the feature `baz` for both classes (in fact, it would suffice to have a feature

with two init state nodes, one for `FOO.baz` and one for `BAR.baz`). Whether this is a feasible approach remains to be evaluated.

Extending CPM+OO for handling simple inheritance is another possibility. Implementing the semantics directly would require representing the inheritance structure in the start graph. When performing queries and commands, rules would then need to first determine the dynamic type of the target object and based on this traverse the inheritance structure and select the correct feature to be applied.

6.6.2 Expanded Types

In CPM+OO, we only support integer and Boolean expanded types. A more general approach would distinguish between expanded types and normal (reference) types. Supporting expanded types is an important step towards full support of SCOOP, but will require considerable effort and requires extending the CPM+OO model, as expanded types are treated different than normal types in the SCOOP semantics [13], and adding them to CPM+OO has implications on existing parts of the model.

6.6.3 Miscellaneous

A number of other features are currently not supported by our toolchain. This includes more exotic features of SCOOP like non-object calls, assigner calls, but also basic features like character and floating point number literals or class invariants. Adding these features to CPM+OO have currently lower priorities as opposed to inheritance and expanded types, but will be considered once the above is implemented properly.

Chapter 7

Case Studies & Evaluation

In the previous two chapters, we discussed the main contribution of this work that allows us to automatically map a subset of SCOOP to CPM+OO start graphs and to verify state-space properties using GROOVE. One part of the motivation behind this work is to provide a "one-click" solution that verifies certain properties—e.g. deciding whether a deadlock can occur—for a given input program written in that SCOOP subset.

In this chapter, we inspect various SCOOP programs as case studies, show how they are translated to a CPM+OO graph using our toolchain, and show the properties we can verify. We provide metrics for the programs to show how the model behaves in various situations and present insights about the gained verification results. We compare our toolchain to CPM and discuss the obtained results, before we close this chapter with an outlook on future work.

We use the following abbreviations to denote program configurations in this chapter.

DP(n, m, {eat, bad_eat}) Dining philosophers with n philosophers and m rounds, as presented throughout this thesis. The last parameter indicates which implementation is used, where eat denotes the correct implementation and bad_eat the implementation that can result in deadlock. This program is presented as a case study in Section 7.2.1.

DS(n, m, o, {bad, good}) Dining savages with pot size n, m savages, and o hunger per savage. The final parameter indicates which implementation is used, where bad is the one that can result in savages being stuck, and good the one that always terminates. This program is presented as a case study in Section 7.2.2.

CS(n) Cigarette smokers problem with n rounds. In this problem, a number of cigarette smokers require different ingredients to build cigarettes, which are provided by a dealer. This program is discussed as a case study in 7.2.3.

SEPC(n) Single-element producer/consumer with n rounds. In this program, a producer and a consumer are created. The producer creates n items that are consumed by the consumer. The producer has a buffer of size 1, which means that `produce` and `consume` calls have to alternate.

Counter(n, m) Counter with n counters and m counts per counter. This is a simple program that spawns a number of counters (n), which simply perform the task of incrementing an integer from 0 to m. While it does not require any synchronisation, it is a small and easy to understand example that showcases SCOOP features.

BS(n, m, o) Barbershop with n customers, m chairs, and o haircuts per customer. This program solves the barbershop problem, where a barber serves several customers. The barber can only cut one customer's hair at a time. Luckily, a waiting room with a number of chairs is available, where customers can wait. Customers can come in the barbershop and take a seat in the waiting room if there is an unoccupied chair. Otherwise, they leave and come back later.

While the cigarette smokers program is our own implementation, the others are taken from the EVE source code repository [22] and adapted to match the input specification of our toolchain.

The above programs make up the main part of the benchmark programs that we used during development of CPM+OO and the translation tool. In addition, we have a number of smaller programs that focus on a certain aspects of the model and translation (e.g. one program provides a wide range of statements involving queries).

7.1 Setup

The values presented in this chapter have been, if not otherwise stated, obtained using the latest revision of the tools, as described in Chapters 4, 5 and 6. We investigate how the system behaves with different setups (e.g. disabling the token passing mechanism for state-space reduction) and use the following two main configurations.

Default Here, all optimisations are turned on, and all rules (in particular error rules) are enabled. Pre- and postconditions are checked as well.

No token optimisation In this configuration, we disable the token optimisation, giving all actions and query operations the same priority. Error rules and pre- and postcondition checking are still enabled.

Values presented in this Chapter represent the median of five (where applicable), and are obtained from a workstation with an Intel Core i7-4810MQ CPU and 16 GB main memory. Runtimes and memory usage are obtained using Java library classes (for CPM+OO measurements) and GNU time 1.7 (for CPM measurements in Table 7.6 only).

7.2 Case Studies

In this section, we take a closer look at three programs.

We start by we revisiting the dining philosophers problem one last time and show how the implementation behaves using our toolchain. We present the implementation and (parts of) the generated start graph, before discussing

evaluation results. In addition, we compare CPM+OO to CPM+OO without token optimisations, and we show how a deadlock can be detected in the bad implementation.

In the second case study, we present the dining savages problem and again inspect two implementations, where the "bad" implementation does not behave as expected. We point out how our toolchain is not able to detect certain undesired behaviours.

Finally, we present the cigarette smokers problem, where we show our implementation and discuss results obtained using both full state-space exploration and LTL formula checking.

7.2.1 Dining Philosophers

In this section, we conclude our running example by presenting an implementation in SCOOP that is in the subset of programs supported by our translation tool. We discuss parts of the start graph and take a closer look at how the program is simulated. Finally, we show how we can detect problems with the implementation, in particular, we show by example how one can detect a deadlock situation.

Source Code

Listings 7.1, 7.2, and 7.3 show the three classes `APPLICATION`, `PHILOSOPHER`, and `FORK` that make up the full working example, with `APPLICATION.make` as the root procedure. The philosopher not only contains a correct implementation of the `eat` feature (which gets called on line 51 in Listing 7.2), but also an implementation called `bad_eat` which, when called instead of `eat`, can result in deadlock. When discussing the analysis we take a look at how CPM+OO handles both variants.

```
1  class
2     APPLICATION
3
4  create
5     make
6
7  feature  -- Initialisation
8
9     i: INTEGER
10    first_fork, left_fork, right_fork: separate FORK
11    a_philosopher: separate PHILOSOPHER
12
13    make
14        -- Create philosophers and forks
15        -- and initiate the dinner.
16      do
17        philosopher_count := 3
18        round_count := 1
19
20        -- Dining Philosophers with `philosopher_count' philosophers
21           and `round_count' rounds.
22
23        from
24          i := 1
25          create first_fork.make
26          left_fork := first_fork
27        until
```

```
27            i > philosopher_count
28        loop
29          if i < philosopher_count then
30            create right_fork.make
31          else
32            right_fork := first_fork
33          end
34          create a_philosopher.make (i, left_fork, right_fork,
                 round_count)
35          launch_philosopher (a_philosopher)
36          left_fork := right_fork
37          i := i + 1
38        end
39      end
40
41 feature {NONE} -- Implementation
42
43    philosopher_count: INTEGER
44        -- Number of philosophers.
45
46    round_count: INTEGER
47        -- Number of times each philosopher should eat.
48
49    launch_philosopher (philosopher: separate PHILOSOPHER)
50        -- Launch a_philosopher.
51      do
52        philosopher.live
53      end
54
55 end
```

Listing 7.1: APPLICATION class.

```
1 class
2   PHILOSOPHER
3
4 create
5   make
6
7 feature -- Initialisation
8
9   make (philosopher: INTEGER; left, right: separate FORK;
        round_count: INTEGER)
10        -- Initialise with ID of `philosopher', forks `left' and `
            right', and for `round_count' times to eat.
11      require
12        valid_id: philosopher > 0
13        valid_times_to_eat: round_count > 0
14      do
15        id := philosopher
16        left_fork := left
17        right_fork := right
18        times_to_eat := round_count
19      ensure
20        id_set: id = philosopher
21        left_fork_set: left_fork = left
22        right_fork_set: right_fork = right
23        times_to_eat_set: times_to_eat = round_count
24      end
25
26 feature -- Access
27
```

```
28     id: INTEGER
29          -- Philosopher's id.
30
31  feature -- Measurement
32
33     times_to_eat: INTEGER
34          -- How many times does it remain for the philosopher to eat?
35
36  feature -- Basic operations
37
38     eat (left, right: separate FORK)
39          -- Eat, having acquired `left' and `right' forks.
40       do
41          -- Eating takes place.
42       end
43
44     live
45       do
46          from
47          until
48             times_to_eat < 1
49          loop
50             -- Philosopher `Current.id' waiting for forks.
51             eat (left_fork, right_fork)
52             --bad_eat
53             -- Philosopher `Current.id' has eaten.
54             times_to_eat := times_to_eat - 1
55          end
56       end
57
58     bad_eat
59          -- Eat, by first picking up `left_fork' (and picking up `
                right_fork'
60          -- in the `pickup_left' call.
61       do
62          pickup_left (left_fork)
63       end
64
65     pickup_left (left: separate FORK)
66          -- After having picked up `left', proceed to pick up `
                right_fork'.
67       do
68          pickup_right (right_fork)
69       end
70
71     pickup_right (right: separate FORK)
72          -- Both forks have been acquired at this point.
73       do
74          -- eating takes place
75       end
76
77  feature {NONE} -- Access
78
79     left_fork: separate FORK
80          -- Left fork used for eating.
81
82     right_fork: separate FORK
83          -- Right fork used for eating.
84
85  invariant
86     valid_id: id >= 1
87
```

```
88  end
```
Listing 7.2: `PHILOSOPHER` class.

```
 1  class
 2     FORK
 3
 4  create
 5     make
 6
 7  feature  -- Initialisation
 8
 9     make
10        do
11        end
12
13  end
```
Listing 7.3: `FORK` class.

In the root procedure (`APPLICATION.make`), we create three philosophers and forks between each pair of adjacent philosophers. The philosophers get initialized with an identifier, the two separate forks they need to pick up, and a round count value, indicating how often they need to eat before terminating. Note that philosophers are started using the call `launch_philosopher ( a_philosopher)`. This is required since `a_philosopher` is of separate type and must be controlled. Passing them as an argument makes it controlled in the called feature, where we are allowed to make the call `philosopher.live`. The code of the philosopher's `make` procedure also uses pre- and postconditions, which we can inspect later through model checking by our verification tools. Note that the preconditions are not wait conditions, as no involved variables are of separate type.

Start Graph

We do not include the complete generated start graph here—with 287 nodes and 789 edges it is too large to print. Instead, we refer to [21], where one can find the CPM+OO GTS and the start graph dining_philosophers_3_philoso-phers_1_round_eat, which represents this instance. We focus on highlighting several interesting parts of the graph and its behaviour under CPM+OO here.

Figure 7.1 shows the `live` procedure of the philosopher (nodes have been rearranged manually for improved readability, but note that the translation tool already performs basic positioning of the graph nodes). The feature starts at node $n5$. The first statement in the feature (`until times_to_eat < 1`) is represented as a pair of Boolean operations with nodes $n9$ and $n11$ and corresponding **Action_Test** nodes that implement the branching. Following the path to $n19$ means that `times_to_eat < 1` and consequently we end up in node $n2$, the final state. Otherwise, the other branch is followed where the loop body is implemented with state nodes $n14$, $n18$, and $n15$. Inside the loop, two actions represent the statements `eat (left_fork, right_fork)` and `times_to_eat := times_to_eat - 1`. State $n15$ leads to two test actions evaluating the `until` part.
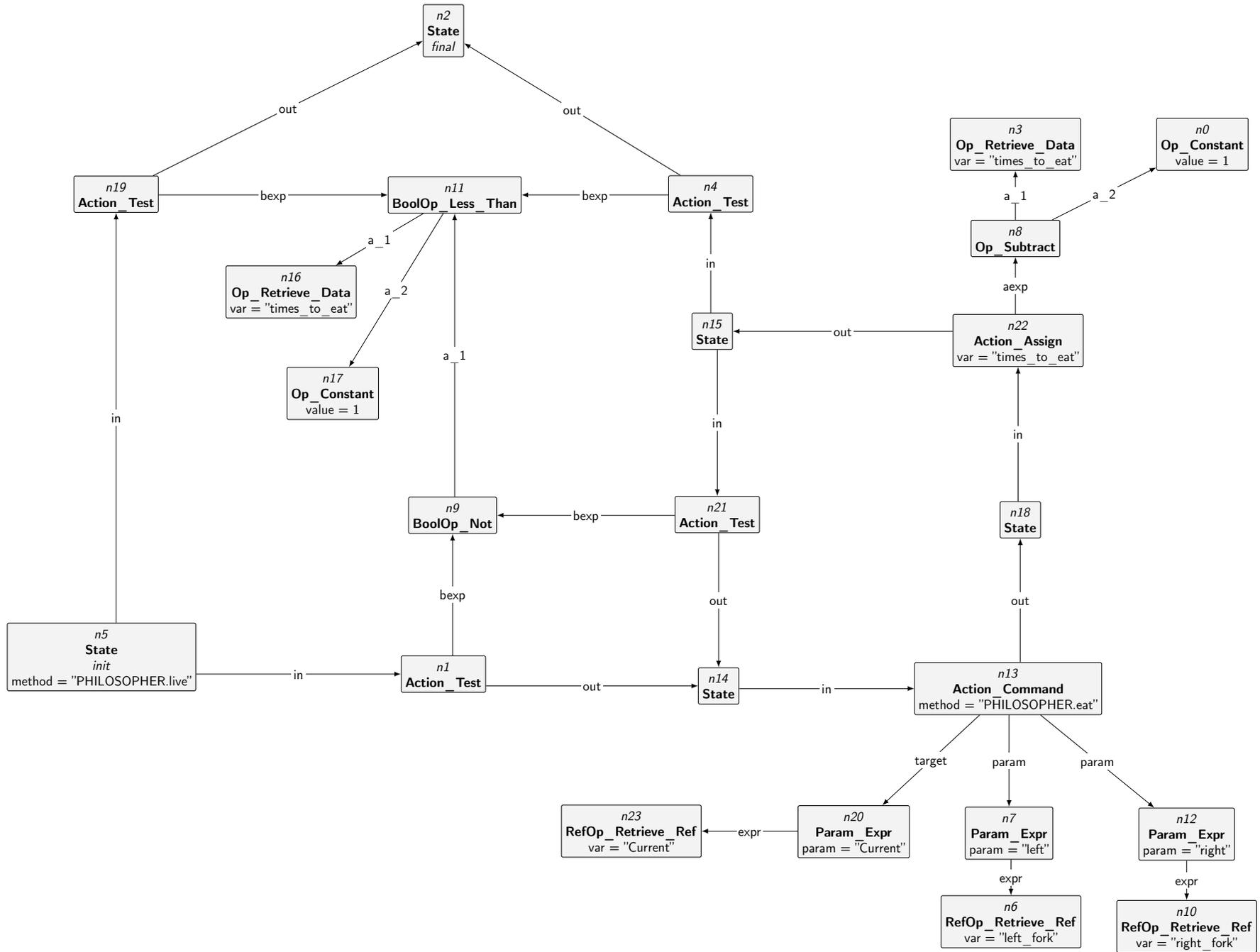
Figure 7.1: Representation of PHILOSOPHER.live in the start graph.

The creation procedure in the philosopher class, shown in Figure 7.2, initializes a number of attributes. In addition, it contains pre- and postconditions which validate the passed arguments and ensure that the attributes have been successfully set. In the graph, the procedure starts at node $n51$ with an init state. First, the handlers of the separate arguments are locked with an **Action_Lock** node. The following test actions represent preconditions. Note that nodes $n43$ and $n6$ have a flag *precondition_fail*. This denotes that these actions represent the path that is taken when a pre- or wait condition fails. This does not necessarily mean that the tested statement is a precondition. Depending on the objects included in the test and whether their handlers are controlled or not, they are either preconditions or wait conditions. This can only be detected at runtime, which is handled by the rule action_Test. When the rule is applied with an action with the *precondition_fail* flag and it turns out that it is in fact a precondition, then the rule creates an **ERROR** node, which has the effect that the system is immediately in a final state, which can be analysed by the postprocessing tools.

Once a processor is in state $n56$, the method body gets executed. When the processor reaches node $n50$, there are two possibilities: Either postcondition checking is disabled, in which case the processor returns to the calling procedure or becomes idle, or postcondition checking is enabled, in which case the edge to $n20$ is followed.

Figure 7.2: PHILOSOPHER.make start graph.

Results

With the start graph discussed earlier, we can now verify different properties of the dining philosophers instance. We are not only interested in whether a deadlock can occur, but we also want to make sure that we never have a call where the target is a void reference, or that postconditions never fail.

We start with the default configuration with optimisations enabled. Table 7.1 shows results with varying numbers of philosophers and rounds (number of times each philosopher eats) for both implementations (`eat` and `bad_eat`). To obtain these results, we used breadth-first search and explored the full state space. Example output from our command-line tool for an instance with the `bad_eat` implementation looks as follows.

```
1 SCOOP sources: dining_philosophers/
     dining_philosophers_2_philosophers_1_round_bad_eat
2 Exploration type: TERMINATION
3 Start graph size (#nodes / #edges): 326 / 494
4 Final graph size (#nodes / #edges): 382 / 650 (0.00, 0.00)
5 Median States: 1282 (0.00)
6 Median Transitions: 1309 (0.00)
7 Median wall clock time: 1.38 (0.57)
8 Median total memory used: 539,321,016 (102,715,198.36)
9 Median new memory since start: 2,618,731 (204,233.22)
10 Min/max result states: 2/2
11 Min/max final states: 2/2
12
13 The simulation generated an error node with label: "Deadlock
     detected".
```

Our tool explores the complete state-space and inspects final graphs. If there are nodes of type **ERROR**, the associated information is fetched and reported. While this kind of output is rather rudimentary, one can use GROOVE to save the offending traces and inspect the program execution to find the root cause of the error.

In cases where no error node is present, our tool first checks whether there are in_method edges in final states, which means that there are processors still executing code while no rule can be applied any more. This indicates that the program is stuck. This situation should not arise, as it means that the program is stuck without a corresponding error rule, which can be either a bug in our implementation, or an error situation we did not define and capture with a rule yet.

Inspecting the numbers in Table 7.1 reveals that we are able to verify deadlock freedom, absence of pre- and postcondition failures (although only a limited number of such statements are in the source code) for the correct implementation with up to seven philosophers, which requires less than 150,000 states and transitions. The runtimes are reasonable, with most instances being evaluated in less than a minute. The numbers for the `bad_eat` implementation are substantially larger. Due to the fact that locking of the forks is not atomic anymore, more interleavings are possible. Even with the smallest instance, this results in an increase of roughly 40% in the amount of states and transitions. With larger instances, the effect is even bigger, with the one with seven philosophers having a state-space of almost 3,000,000 states. The runtime of roughly 85 minutes is substantially longer than the runtime of the corresponding correct implementation.

n	i	Impl.	States	Transitions	Time [stddev] (s)	Memory [stddev] (GB)
2	1	eat	962	1019	1.26 [0.49]	0.59 [0.09]
3	1		2976	3134	3.08 [0.70]	0.67 [0.18]
3	2		7974	8662	8.23 [0.67]	0.96 [0.20]
3	3		16,208	17,836	15.89 [0.70]	1.96 [0.41]
3	5		45,264	50,410	45.98 [0.87]	3.16 [0.72]
4	1		8326	8720	9.38 [0.84]	1.29 [0.27]
5	1		21,814	22,748	26.18 [0.87]	2.98 [0.91]
6	1		54,638	56,788	75.14 [1.03]	4.23 [0.27]
7	1		132,518	137,372	202.84 [3.82]	5.51 [0.41]
2	1	bad_eat	1358	1423	1.70 [0.44]	0.49 [0.07]
3	1		6528	6888	6.69 [0.43]	0.99 [0.19]
3	2		21,130	22,372	22.12 [1.31]	1.91 [0.42]
3	3		47,859	50,759	48.81 [1.69]	3.82 [0.76]
3	5		150,471	159,855	155.88 [3.26]	5.17 [0.24]
4	1		31,105	32,961	37.89 [1.20]	3.11 [0.88]
5	1		144,891	154,116	187.63 [2.51]	5.25 [0.24]
6	1		662,009	706,430	963.65 [10.73]	9.36 [1.04]
7	1		2,972,519	3,181,087	5001.01 [21.04]	12.46 [0.10]

Table 7.1: Results from full state-space exploration of various instances of the dining philosophers program, where n denotes the number of philosophers and i the number of times each philosopher eats.

If one is only interested in detecting whether a deadlock can occur or not, an alternative approach is to use LTL formula exploration instead of full state-space exploration. In this approach, one can instruct GROOVE to try and find a counterexample to an LTL formula. To detect a deadlock, we can use the formula ! F error_deadlock. Table 7.2 shows the corresponding results. As we can see, the numbers of explored states and transitions for the correct implementation do not differ from Table 7.1, as, in order to prove that no counterexample exists, one has to explore the full state-space. For the bad implementation on the other hand, the number of explored states and transitions are substantially smaller. While this may seem like an improvement at first, taking a look at the runtimes reveals that checking the formula comes at a cost. While the smaller instances of the correct implementation take roughly the same time in both cases, using LTL exploration takes longer with the larger instances. In the bad implementation, finding a counterexample is faster for small instances, but with larger ones, it takes longer to check the formula, even though fewer states are explored. In the case with 6 philosophers, finding a counterexample requires (on average) only 441,416 states compared to the 662,009 states of the full state-space. Nevertheless, finding the counterexample takes more than twice the time as compared to exploring the full state-space.

Disabling Optimisations The above numbers are quite promising, as we not only can verify a minimal dining philosophers program, but also instances with a larger number of involved processors and rounds. Reducing the runtimes has helped us immensely during development, as we can test changes made to the system almost instantly, where we previously had to wait several minutes for a result. The story is quite different if we look at earlier revisions of CPM+OO. The feature that has the biggest impact is the optimisation using the execution token which marks the processor that is allowed to execute sequential actions, and where the system only processes separate actions once no processor can make sequential progress any more. If we disable this functionality, we obtain the

n	i	Impl.	States	Transitions	Time [stddev] (s)	Memory [stddev] (GB)
2	1	eat	962	1019	1.01 [0.30]	0.64 [0.14]
3	1		2976	3134	3.21 [0.78]	0.81 [0.15]
3	2		7974	8662	8.20 [0.74]	1.21 [0.26]
3	3		16,208	17,836	17.18 [0.98]	2.09 [0.42]
3	5		45,264	50,410	60.23 [1.60]	3.65 [0.56]
4	1		8326	8720	8.94 [0.75]	1.36 [0.29]
5	1		21,814	22,748	28.35 [0.88]	3.61 [0.85]
6	1		54,638	56,788	102.30 [1.79]	4.25 [0.23]
2	1	bad_eat	828	837	0.92 [0.55]	0.46 [0.09]
3	1		3549	3686	3.59 [0.87]	0.80 [0.16]
3	2		4718	4891	4.52 [1.44]	1.02 [0.20]
3	3		3950	4080	4.12 [2.83]	0.94 [0.18]
3	5		2192	2218	2.12 [4.09]	1.25 [0.24]
4	1		18,549	19,412	22.03 [0.68]	2.96 [0.78]
5	1		85,059	89,623	174.54 [20.11]	4.23 [0.05]
6	1		441,416	467,285	2150.79 [112.63]	5.97 [0.25]

Table 7.2: Exploration of LTL formula ! F error_deadlock. Exploration with more than six philosophers has been aborted after more than 2,500,000 states.

n	i	Impl.	States	Transitions	Time [stddev] (s)	Memory [stddev] (GB)
2	1	eat	15,480	18,265	15.43 [0.90]	1.97 [0.32]
3	1		252,112	304,409	328.91 [19.60]	6.15 [0.58]
3	2		711,640	877,576	769.02 [9.55]	9.18 [1.13]
3	3		1,526,582	1,903,627	1690.66 [29.32]	12.56 [1.85]
2	1	bad_eat	21,236	24,417	20.53 [0.94]	2.63 [0.49]
3	1		425,983	499,660	487.62 [14.94]	7.87 [0.78]
3	2		1,445,738	1,710,118	1579.87 [19.29]	12.52 [1.49]
3	3		3,417,959	4,059,490	4065.24 [169.68]	12.60 [0.19]

Table 7.3: Results from verifying the correct implementation without token optimisation.

numbers presented in Table 7.3. Verifying the instance with three philosophers and a single round already results in more than 250,000 states and 300,000 transitions, a huge difference compared to the numbers with the optimisation turned on.

The difference can be explained with the fact that without the token mechanism, large chunks of the program get simulated over and over again in different interleavings without affecting the outcome. For example, consider the situation where a processor is in state $n5$ of Figure 7.1 and has already evaluated the arguments to the test action nodes. Without the token optimisation, at this point all other processors could simulate their states until they are finished. Another execution plan would first advance our processor to node $n14$ (assuming $n9$ has been evaluated to true), before simulating the remaining processors all over again. There is no value in considering both variants, as the outcome of the test action, which is a purely local step, does not depend on any outside properties of the system (in particular, it does not depend on the states of other processors). With the token mechanism, we force the system to take one particular path in these situations and only allow branching at points where processors can potentially interact with each other and where different outcomes can originate.

Detecting Deadlocks   The "eat" implementation behaves as expected. The simulation is unable to find any issues with it, in particular, we are not able to find a situation where the program deadlocks. In this section though, we inspect the alternative implementation of the philosopher's behaviour (i.e. the `bad_eat` method in Listing 7.2). Since we want to find out whether a deadlock can occur or not, we can do so by using the LTL formula ! F error_deadlock, which states that, starting from the start graph, there is no future state where the rule error_deadlock matches. GROOVE explores the state-space and reports whether a counterexample exists for the formula. If so, we have a situation where a deadlock occurs and we can inspect the trace that leads from the start graph to that particular state.

When using the `bad_eat` implementation, we can in fact find counterexamples to the formula. Figure 7.3 shows an excerpt of such a state with the involved processors, locks, and states. Both processors are in the `pickup_right` command and hold their left fork (which is the other one's right fork). Each processor holds the lock to the processor the other one is waiting for. As a result, no processor can make progress, and we have a deadlock situation which is detected by the error_deadlock rule.
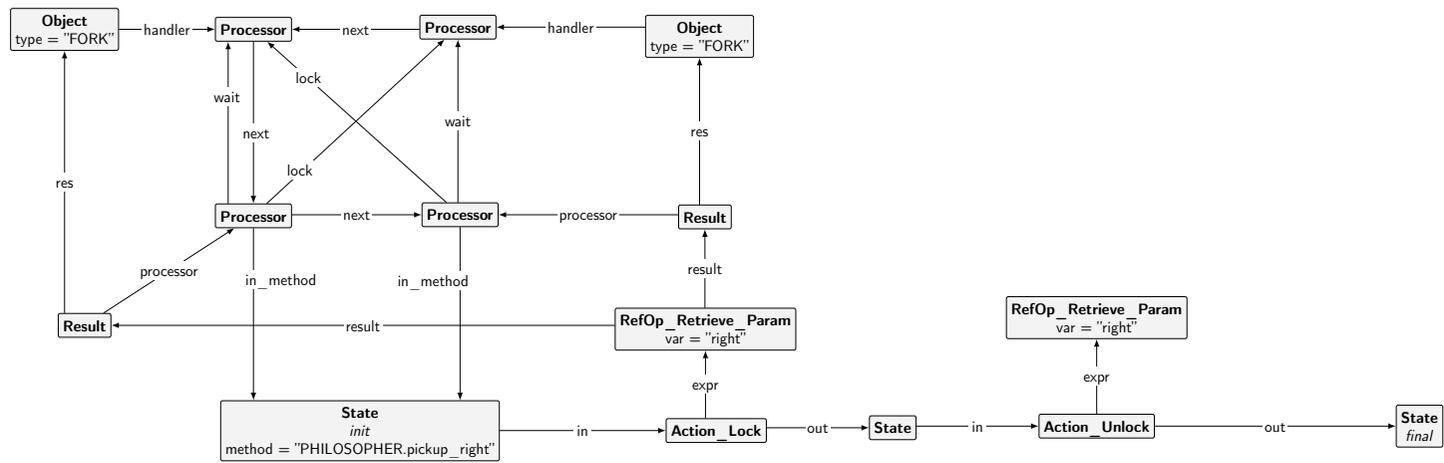
Figure 7.3: Deadlock situation with 2 philosophers.

7.2.2 Dining Savages

Our second case study is the dining savages problem. The premise is that there are a number of savages that share a single pot that contains their food. A cook can fill the pot, and each savage can get servings from it. Since the pot is rather small, the number of servings is limited and only one savage can get a serving at a time. If a savage is trying to get a serving when the pot is empty, he notifies the cook to fill it up again, waits until the cook does is job, and then gets his serving.

Source Code

Our implementation of the program consists of four classes. Apart from the `APPLICATION` class that initializes and starts the system, there are classes for representing the cook, a savage, and the pot. In our implementation, we have three configuration variables, namely the pot size (number of servings the pot can hold), the number of savages, and the hunger of a savage (which is the number of servings a savage is going to take before terminating). The program first creates all objects and then launches the savages. Listing 7.4 shows relevant code of the savage class. During the lifetime of a savage, it executes the `live` feature which is a simple loop that executes `step` a number of times. In a step, a savage calls `fill_pot` which notifies the cook to fill the pot, if necessary. The program continues, since `fill_pot` can return even if the pot is empty, as the command that can get called in the routine body is asynchronous. Afterwards, `get_serving_from_pot` gets called. Since—in case the pot was empty or has become empty in the meantime—we can not be sure whether the cook already filled the pot, we use the wait condition `not` `my_pot.is_empty`. This ensures that the savage gets the serving from a non-empty pot. Once the wait condition is satisfied, the savage has exclusive access to the pot, which means that it is impossible for the pot to become empty before the savage can call `my_pot.get_meal`. The final command in a step of the savage is eating, which simply decreases the hunger value to avoid having an infinite loop in the `live` procedure.

   While the original implementation defines `step` without an argument, we pass the pot to this feature in our adapted implementation. We do this to avoid processors being stuck in a wait condition that never gets fulfilled. Consider the instance where the pot can hold one serving and two savages want to eat only once. Without passing the pot to the `step` feature call, the following sequence can occur.

- Savage 1 calls `fill_pot`, sees that the pot is full and therefore does not ask the cook to fill it.

- Savage 2 calls `fill_pot`, sees the same, and does not ask the cook either.

- Savage 1 calls `get_serving_from_pot`, passes the wait condition, and returns. The pot is now empty. Savage 1 has finished its loop and does not execute anything any more.

- Savage 2 calls `get_serving_from_pot`, but is stuck in the wait condition, as the pot is now empty. Since Savage 1 has finished, the pot will never get filled again and Savage 2 is stuck forever.

By passing the pot as an argument to the `step` routine, we ensure that for one savage in a single step, all operations involving the pot are executed without interleaving requests from another savage. This makes the above interleaving impossible and as a result, savages can not get stuck any more. Note that the condition in `get_serving_from_pot` is now a precondition, as the request queue of the handler of the pot is already locked when `get_serving_from_pot` gets called. We call this the "good" implementation, but also take a look at the implementation where we do not pass the pot to `step`, which we call "bad".

```eiffel
 1  feature {NONE} -- Access
 2    step (a_pot: separate POT)
 3        -- Perform a savage's tasks.
 4      do
 5        fill_pot (a_pot, cook)
 6        get_serving_from_pot (a_pot)
 7        eat
 8      end
 9
10    over: BOOLEAN
11      do
12        Result := hunger = 0
13      end
14
15  feature {NONE} -- Implementation
16    fill_pot (my_pot: separate POT; my_cook: separate COOK)
17        -- Fills pot if it's empty.
18      do
19        if my_pot.is_empty then
20          my_cook.cook (my_pot)
21        end
22      end
23
24    get_serving_from_pot (my_pot: separate POT)
25        -- Gets the meal from the pot
26      require
27        not my_pot.is_empty
28      do
29        my_pot.get_meal
30      end
31
32    eat
33        -- Eat the meal.
34      require
35        hunger > 0
36      do
37        hunger := hunger - 1
38      end
39
40  feature {NONE}
41
42    id: INTEGER
43    pot: separate POT
44    cook: separate COOK
45
46    hunger: INTEGER
47
48  feature -- Process behaviour
49    live
50      do
51        from
52        until
```

```
53          over
54        loop
55          step (pot)
56        end
57      end
58  end
```

Listing 7.4: Savage implementation. escapechar

Parts of the source code of the cook is shown in Listing 7.5. In the **require** block of the **cook** feature, we use another wait condition to make sure that the pot is in fact empty when the feature body gets executed.

```
1  feature
2
3      do_cooking
4          -- Wrapper call to control pot.
5        do
6          cook (pot)
7        end
8
9      cook (a_pot: separate POT)
10          -- Fill the pot.
11        require
12          a_pot.is_empty
13        do
14          a_pot.fill
15        ensure
16          a_pot.is_full
17        end
18  end
```

Listing 7.5: Cook implementation.

Results

Table 7.4 lists results obtained with CPM+OO for a number of instances of the dining savages program. The instances range from two to four savages and two to six total calls to the `get_serving_from_pot` routine. Like in the dining philosophers case, the number of involved processors has the largest impact. Even if we lower the number of times a savage eats in the last instance (with 4 savages), the state-space is by far the biggest in both implementations. This is no surprise, as with more processors, the number of synchronisation points (i.e. situations during the execution where multiple non-separate actions or queries can be performed) increases, and individual synchronisation points may include more processors, resulting in more branching in the LTS.

Comparing both implementations paints a similar picture as what we have seen in the dining philosophers example. In the "bad" implementation, we perform less restrictive locking, thus allow more possible interleavings. While the impact of the smaller instances is negligible, it becomes obvious with the larger ones. In the instance with 4 savages, the "bad" implementation takes about three times longer than the "good" implementation. Note that we do full state-space exploration here, and our tool does not report an issue with both implementations, i.e. no **ERROR** nodes get generated. While savages can get stuck in wait conditions in the "bad" implementation, these situations are not deadlock situations. A processor may never proceed past the wait condition, but

n	m	o	Impl.	States	Transitions	Time [stddev] (s)	Memory [stddev] (GB)
1	2	1	good	3365	3472	4.91 [0.54]	0.70 [0.09]
4	2	2		5710	5923	8.58 [0.65]	0.99 [0.18]
2	2	2		6121	6340	9.28 [0.53]	0.88 [0.16]
2	3	2		66,592	70,044	124.51 [1.63]	4.51 [0.51]
2	4	1		155,578	165,157	329.14 [5.04]	5.75 [0.42]
1	2	1	bad	4193	4396	7.07 [0.66]	0.78 [0.13]
4	2	2		8479	8999	13.35 [0.62]	1.11 [0.21]
2	2	2		9147	9668	14.39 [0.82]	1.26 [0.26]
2	3	2		178,493	191,810	346.94 [2.16]	5.93 [0.39]
2	4	1		431,900	466,498	962.53 [12.65]	8.76 [0.94]

Table 7.4: Results of various instances of the dining savages program.

it can make progress in the sense that the wait condition is checked over and over again (as requests are generated for and executed by the handler of the pot). The target processor is idle and can execute the requests from the stuck processor. In the LTS, this results in a local cycle of states, where there is no path that "breaks out" from this cycle. Currently, we are unable to detect such situations, and whether it is possible to detect them using LTL or CTL formulae remains to be investigated in future work.

7.2.3 Cigarette Smokers Problem

In our final case study, we implement and evaluate the cigarette smokers problem. In this problem, there are three cigarette smokers wanting to build cigarettes and smoke them. Their problem is that each one has only one of the required three ingredients, namely tobacco, matches, or papers. Thankfully, a dealer is available that has an infinite amount of each ingredient. The dealer randomly makes two of them available at a time, allowing the smoker with the third ingredient to retrieve them and then build and smoke a cigarette.

The original premise, which we borrow from [4], states that both the dealer's supply as well as the smoker's desire to smoke are infinite. We change this in order to get a program that terminates. In particular, we now require that all smokers only retrieve ingredients and smoke n times. In addition, the dealer puts out each distinctive pair of ingredients exactly n times. As a result, nobody is stuck waiting, as once the dealer has put out every pair n times, he can go home, and all smokers are satisfied as they were able to build and smoke a cigarette n times.

Source Code

Our implementation of the problem consists of three main classes, `DEALER`, `CLIENT`, and `INGREDIENT_PAIR`. The dealer is a simple class resembling a semaphore and is used to make sure that no two pairs are available at the same time (which would imply that all three ingredients are available, which is not allowed in the problem statement). Listing 7.6 shows the full source code of the `DEALER` class. Instead of using a class to represent individual ingredients, we use one to represent pairs of ingredients. Since we want a limited amount of each pair, we can use this representation to force each pair a fixed number of times. Ingredient pairs (Listing 7.7) are created with a separate dealer, and the `put_out` feature is called after creation. A pair then, if the dealer is not busy,

```
1  class
2    DEALER
3
4  create
5    make
6
7  feature
8    make
9      do
10        is_available := true
11      end
12
13    set_available
14      do
15        is_available := true
16      end
17
18    set_busy
19      do
20        is_available := false
21      end
22
23    is_available: BOOLEAN
24  end
```

Listing 7.6: `DEALER` class.

puts itself out, ready to be consumed by a client (Listing 7.8). Once a pair is consumed via the `consume` feature, it either terminates, or tries to put itself out again. A client gets initialized with a pair of ingredients, and simply calls `consume` n times which is blocked until its ingredient pair is actually out.

With pairs of ingredients as separate objects, we introduce the randomness specified in the problem statement. The dealer serves as semaphore that can be occupied by one pair at a time, ensuring that no two pairs are out at the same time. The source code is, thanks to expressive wait conditions, rather simple and clear.

Results

The generated start graph of the cigarette smokers program consists of more than 400 nodes and 1100 edges. We are confident that our implementation works correctly, and therefore it is no surprise that no **ERROR** states are generated when verifying instances of this program. Table 7.5 shows results for various instances of the program, where we varied the number of times each smoker and ingredient pair execute their corresponding loops. In the top half of the table, we explore the full state-space and report on any **ERROR** nodes generated in final states. While the state-space grows quickly, we nevertheless are able to verify in a reasonable amount of time that instances with up to 5 rounds do not exhibit any of the bad properties we are looking for.

In the lower half, we used the LTL formula !F error_deadlock to find counterexamples for a deadlock error rule. Since we did not find such an error in the full state-space exploration above, it is no surprise that the exploration does not find such a counterexample when using LTL property checking either. We can observe once again that formula checking comes at a cost: all instances take

```
1  class
2     INGREDIENT_PAIR
3
4  create
5     make
6
7  feature
8     make (an_id: INTEGER; a_count: INTEGER; a_dealer: separate DEALER
          )
9        do
10          id := an_id
11          dealer := a_dealer
12          is_out := false
13          capacity := a_count
14       end
15
16    put_out
17       do
18          put_out_with_dealer (dealer)
19       end
20
21    put_out_with_dealer (a_dealer: separate DEALER)
22       require
23          a_dealer.is_available
24       do
25          a_dealer.set_busy
26          is_out := true
27       end
28
29    is_out: BOOLEAN
30
31    consume
32       do
33          consume_with_dealer (dealer)
34
35          capacity := capacity - 1
36          if capacity > 0 then
37             put_out
38          end
39       end
40
41    consume_with_dealer (a_dealer: separate DEALER)
42       do
43          a_dealer.set_available
44       end
45
46    dealer: separate DEALER
47    id: INTEGER
48    capacity: INTEGER
49       -- total times the pair is available
50
51  end
```

Listing 7.7: `INGREDIENT_PAIR` class.

```eiffel
class
  CLIENT

create
  make

feature
  make (an_id, a_count: INTEGER; a_ingredients: separate
        INGREDIENT_PAIR)
    do
      id := an_id
      count := a_count
      ingredients := a_ingredients
    end

  start
    local
      i: INTEGER
    do
      from
        i := 0
      until
        i = count
      loop
        consume
        i := i + 1
      end
    end

  consume
    do
      consume_with_pair (ingredients)
    end

  consume_with_pair (a_pair: separate INGREDIENT_PAIR)
    require
      a_pair.is_out
    do
      a_pair.consume
    end

  id: INTEGER
  count: INTEGER
  ingredients: separate INGREDIENT_PAIR

end
```

Listing 7.8: CLIENT class.

Rounds	Type	States	Transitions	Time [stddev] (s)	Memory [stddev] (GB)
1	default	69,130	75,013	191.99 [4.56]	4.64 [0.25]
2		269,497	291,593	755.23 [16.03]	6.24 [0.63]
3		602,402	649,519	1918.49 [27.66]	8.57 [0.73]
4		1,101,193	1,184,059	3084.29 [46.23]	9.74 [1.27]
5		1,799,218	1,930,481	5007.15 [84.28]	12.45 [1.06]
1	deadlock	69,130	75,013	233.25 [2.62]	4.65 [0.14]
2		269,497	291,593	1376.87 [24.35]	5.66 [0.25]
3		602,402	649,519	4025.06 [54.42]	5.57 [0.10]

Table 7.5: Results of various instances of the cigarette smokers program.

more time in the lower half of the table compared to their counterparts in the upper half. While we can argue that in these cases we gain more information and are faster when exploring the full state space, it is important to note that LTL checking has value as well. In particular, when looking at instances where we are no longer able to explore the full state-space, i.e. when having to rely on bounded model checking, looking for counterexamples of properties is the only way to gain any valuable information.

7.3   Comparison with CPM

In this section, we take a look at how various variants of our models perform. In CPM+OO, we introduced a number of abstractions and except considerable overhead in the form of a larger state-space, as compared to CPM. To reduce the state-space size in CPM+OO, we used several optimisations, in particular we used more quantifiers in certain rules and we introduced the token execution optimisation.

Since there is currently no translation tool that can generate start graphs for CPM, we have to create start graphs by hand. This makes translation of real-world examples tedious and is error-prone. As a result, there are only a limited number of start graphs for use with CPM available. Most notably, we have start graphs for the dining philosophers problem with both implementations, as well as a start graph for single-element producer/consumer. It is important to note that start graphs between CPM and CPM+OO for the same program do differ substantially, making the comparison more difficult. The graphs for CPM are simpler in most regards: there are no local calls, no evaluation of call targets (instead it is directly specified by the name of an attribute), and other abstractions introduced in CPM+OO are missing as well. Nevertheless, it is interesting to compare results of those two models. Additional abstractions in CPM+OO have led to less direct action applications, which resulted in additional rules and interleavings, but using optimisations has helped reduce the state-space again.

Table 7.6 shows results obtained for the dining philosophers and the single-element producer/consumer examples with three models, namely CPM, CPM+OO, and CPM+OO without token optimisation. In the case of CPM, we use start graphs translated and adapted by hand. In the other cases, we use our SCOOP reference implementations and generate start graphs with our translation tool.

Comparing the numbers of CPM to the ones of CPM+OO without optimisations shows that the size of the state-space of the latter exceeds the size of the

state-space generated with CPM. The DP(3, 2, bad_eat) instance takes around 27 minutes to verify using CPM+OO without the token optimisation, whereas the same instance verified with CPM takes less than 4 minutes. Compared to CPM+OO without optimisations, CPM performs better across all instances. We explain this with the above arguments, namely that CPM+OO increases complexity and adds more abstractions that require additional computations. In addition, the start graphs of the CPM instances have been generated by hand and are optimised for these examples.

Fortunately, the token optimisation has a huge impact in the performance of CPM+OO. When enabling the optimisations, we can verify each instance in under 30 seconds. Not only does this outperform CPM+OO without optimisations by a huge margin, but it is also considerably faster and generates smaller state-spaces than CPM with optimised start graphs.

We realise that comparing the models using start graphs that differ as much as they do is not optimal. Still, in our opinion this comparison gives a good impression of the effect of optimisations and shows that—although CPM+OO is inherently more complex than CPM—we manage to not only preserve the size of generated state-spaces, but thanks to optimisations we are even able to produce smaller state-spaces focusing on the synchronization points of the programs.

Model	Program	States	Transitions	Time [stddev] (s)	Memory [stddev] (GB)
CPM	DP(2, 1, bad_eat)	3923	4990	8.89 [1.16]	1.11 [0.08]
	DP(3, 1, bad_eat)	41,347	54,749	71.27 [1.68]	3.51 [0.56]
	DP(3, 2, bad_eat)	138,059	180,173	231.29 [11.12]	4.93 [0.04]
	DP(2, 1, eat)	3404	4281	8.06 [0.19]	1.10 [0.01]
	DP(3, 1, eat)	32,155	41,793	56.38 [0.86]	2.70 [0.87]
	DP(3, 2, eat)	104,131	133,304	185.22 [3.95]	4.80 [0.06]
	SEPC(5)	17,864	21,763	37.76 [1.51]	2.81 [0.64]
	SEPC(20)	76,949	93,718	148.51 [5.73]	4.71 [0.09]
CPM+OO (no token)	DP(2, 1, bad_eat)	21,236	24,417	20.53 [0.94]	2.63 [0.49]
	DP(3, 1, bad_eat)	425,983	499,660	487.62 [14.94]	7.87 [0.78]
	DP(3, 2, bad_eat)	1,445,738	1,710,118	1579.87 [19.29]	12.52 [1.49]
	DP(2, 1, eat)	15,480	18,265	15.43 [0.90]	1.97 [0.32]
	DP(3, 1, eat)	252,112	304,409	328.91 [19.60]	6.15 [0.58]
	DP(3, 2, eat)	711,640	877,576	769.02 [9.55]	9.18 [1.13]
	SEPC(5)	106,526	126,392	152.03 [3.30]	4.67 [0.22]
	SEPC(20)	462,221	549,527	685.70 [13.13]	7.07 [0.90]
CPM+OO	DP(2, 1, bad_eat)	1358	1423	1.70 [0.44]	0.49 [0.07]
	DP(3, 1, bad_eat)	6528	6888	6.69 [0.43]	0.99 [0.19]
	DP(3, 2, bad_eat)	21,130	22,372	22.12 [1.31]	1.91 [0.42]
	DP(2, 1, eat)	962	1019	1.26 [0.49]	0.59 [0.09]
	DP(3, 1, eat)	2976	3134	3.08 [0.70]	0.67 [0.18]
	DP(3, 2, eat)	7974	8662	8.23 [0.67]	0.96 [0.20]
	SEPC(5)	2338	2412	3.70 [0.90]	0.61 [0.10]
	SEPC(20)	9088	9372	13.28 [0.79]	0.98 [0.18]

Table 7.6: Comparison of performance of CPM, CPM+OO, and CPM+OO without token optimisation.

7.4   Scalability and Future Work

So far, we have only considered input programs that resulted in state-spaces that can be fully explored with our toolchain. In our development environment, we have generated LTSS with up to 4,000,000 states, at which point we ran out of memory. The number of states that can be explored depends on a number of variables though. For example, the chosen exploration strategy can be a factor, as well as the model and start graph. With larger programs where full state-space exploration is not feasible any more, one can consider doing bounded verification, where one explores only parts of the state-space. It is important to note that with bounded verification, it is only possible to search for instances of errors, but there is no guarantee that an error can be found, and the absence of errors can not be proved.

GROOVE offers a range of different exploration strategies. So far, we have used breadth-first-search and LTL exploration. For larger state-spaces, this may not be the optimal choice. For example, when searching the state-space with breadth-first search, one can only reach a particular depth. This may be undesirable, e.g. when the synchronization points that may result in a deadlock occur only later in a program. By searching for counterexamples with depth-first search instead, one may be able to actually reach such synchronization points. Of course, with this approach, not all branches are explored and one might as well miss the ones that result in a deadlock. GROOVE also offers other exploration strategies, such as random linear exploration, where exactly one path is followed per state, or conditional exploration with restrictions on the number of edges and nodes in a state. Custom exploration strategies tailored to CPM and CPM+OO should also be considered.

A thorough investigation regarding bounded exploration and using various exploration strategies to gain confidence in the obtained results is out of scope of this thesis and remains to be done in future work.

Chapter 8

Conclusion

In this chapter, we conclude the thesis. We start with a review of the research hypothesis and summarise our efforts and contributions, before we close the thesis with some final words on future work.

8.1 Contributions

In Section 1.2, we stated the following research hypothesis.

> A subset of valid SCOOP programs can be modelled using a graph transformation system. These programs can, without modification of the source code, be automatically translated to input graphs for the transformation system. Using verification by model checking, it is possible to verify a number of properties such as absence of deadlock or absence of precondition violations for a given input program.

In our opinion, this thesis satisfies the hypothesis with the following contributions.

In Chapters 4 and 5, we described formal models, implemented in GROOVE, that can be used to simulate a subset of SCOOP programs. In the former, we discussed CPM, which focuses on the concurrency features of SCOOP. In the latter, we extend CPM by adding object-oriented features from SCOOP to obtain the CPM+OO model. By careful (informal) reasoning in individual steps, we were able to preserve confidence in the correctness and completeness of the model.

In Chapter 6, we presented a simple compiler that takes a subset of valid SCOOP programs as input and generates input graphs for our formal model. While we are not able to support the complete SCOOP language, we were able to translate a number of real-world concurrent example programs, as later discussed in Chapter 7. In addition, by embedding the GROOVE binaries, we also created a simple command-line interface that can be used to verify SCOOP programs matching the input specification with one single command. This supports the research hypothesis, as the tool works on unmodified SCOOP code.

We evaluated our approach in Chapter 7 with several case studies. We investigated a number of implementations of problems suited for demonstrating concurrent programming, such as the well-known dining philosophers problem, and we have shown how our translation tool and the models behave. We discussed

various aspects of the current version of the CPM+OO model, but also presented the effects of state-space optimisations and a comparison to CPM. We have seen that our toolchain can verify properties like absence of deadlock or absence of precondition violations for the inspected programs, which supports the research hypothesis.

8.2 Future Work

With our tools and model, we are able to translate a number of SCOOP programs and can verify certain properties, such as the absence of deadlock scenarios. While our input programs already use a number of object-oriented features of SCOOP, we are a long way from supporting the complete language. In particular, both the translation tool and the model lack support for inheritance. Our focus in the future is to extend the model and compiler to support a larger subset of SCOOP programs.

We provide a simple tool that works on SCOOP source code and prints out verification results with a single command. The tool outputs simple messages that state which errors could be found. Ideally, the tool should be integrated with the EVE [22] integrated development environment, which combines a number of other verification and analysis approaches. Ultimately, the goal should be to provide a GUI interface that is intuitive and easy to use. The output should be more verbose, extracting more information about the situations that occur (e.g. stating which features processors are executing when a deadlock is detected).

In this thesis, we have been using sample input programs that generate small state-spaces that can be fully explored within minutes or hours. A thorough investigation and evaluation of our work with respect to larger programs and bounded verification remains to be done.

While this thesis focuses on the SCOOP model, it may be possible to adapt our approach to other concurrency languages and models, such as *Grand Central Dispatch* (GCD) [7]. An evaluation remains as future work.

Glossary

ANTLR ANother Tool for Language Recognition.

CPM+OO CPM with Object Orientation.

CPM Concurrent Processor Model.

CTL Computational Tree Logic.

coreSCOOP A subset of scoop focusing on concurrency features.

DPO Double-Pushout.

EBNF Extended Backus-Naur Form.

GCD Grand Central Dispatch.

GPS Graph Production System.

GROOVE GRaphs for Object-Oriented VErification.

GTS Graph Transformation System.

GXL Graph eXchange Language.

LTL Linear Temporal Logic.

LTS Labelled Transition System.

SCOOP Simple Concurrent Object-Oriented Programming.

SPO Single-Pushout.

List of Figures

List of Listings

List of Tables

Bibliography

[1] Phillip J. Brooke, Richard F. Paige, and Jeremy L. Jacob. "A CSP model of Eiffel's SCOOP". In: *Formal Aspects of Computing* 19.4 (2007), pp. 487–512.

[2] PhillipJ. Brooke, RichardF. Paige, and JeremyL. Jacob. "A CSP model of Eiffel's SCOOP". English. In: *Formal Aspects of Computing* 19.4 (2007), pp. 487–512. ISSN: 0934-5043.

[3] Georgiana Caltais and Bertrand Meyer. "Coffman deadlocks in SCOOP". In: *CoRR* abs/1409.7514 (2014).

[4] Allen B. Downey. *The little book of semaphores*. Accessed April 10, 2015. URL: http://greenteapress.com/semaphores/.

[5] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer, 2006.

[6] A. H. Ghamarian et al. *Modelling and Analysis Using GROOVE*. Technical Report TR-CTIT-10-18. Enschede: Centre for Telematics and Information Technology University of Twente, Apr. 2010.

[7] *Grand Central Dispatch (GCD) Reference*. https://developer.apple.com/library/mac/documentation/Performance/Reference/GCD_libdispatch_Ref/index.html. Accessed April 10, 2015.

[8] Alexander Heußner, Christopher M. Poskitt, Claudio Corrodi, and Benjamin Morandi. "Towards Practical Graph-Based Verification for an Object-Oriented Concurrency Model". In: *Proc. Graphs as Models (GaM 2015)*. Electronic Proceedings in Theoretical Computer Science. To appear. 2015.

[9] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning About Systems*. Cambridge University Press, 2004.

[10] Bertrand Meyer. *Object-oriented Software Construction (2Nd Ed.)* Prentice-Hall, Inc., 1997.

[11] Bertrand Meyer. "Systematic Concurrent Object-oriented Programming". In: *Commun. ACM* 36.9 (Sept. 1993), pp. 56–80.

[12] Bertrand Meyer. *Touch of Class: Learning to Program Well with Objects and Contracts*. 1st ed. Springer, 2009.

[13] Benjamin Morandi. "Prototyping a Concurrency Model". Doctoral dissertation. ETH Zürich, 2014.

[14] Benjamin Morandi, Mischael Schill, Sebastian Nanz, and Bertrand Meyer. "Prototyping a Concurrency Model". In: *13th International Conference on Application of Concurrency to System Design, ACSD 2013, Barcelona, Spain, 8-10 July, 2013*. 2013, pp. 170–179.

[15] P. Nienaltowski. "Practical framework for contract-based concurrent object-oriented programming". Doctoral dissertation. ETH Zürich, 2007.

[16] Jonathan S. Ostroff, Faraz Ahmadi Torshizi, Hai Feng Huang, and Bernd Schoeller. "Beyond contracts for concurrency". In: *Formal Aspects of Computing* 21.4 (2009), pp. 319–346.

[17] Christopher M. Poskitt. "Verification of Graph Programs". PhD thesis. University of York, 2013.

[18] A. Rensink. "Isomorphism Checking in GROOVE". In: *Graph-Based Tools (GraBaTs), Natal, Brazil*. Vol. 1. Electronic Communications of the EASST. European Association of Software Science and Technology, Sept. 2007.

[19] A. Rensink. "The GROOVE Simulator: A Tool for State Space Generation". In: *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*. Vol. 3062. LNCS. Berlin: Springer Verlag, 2004, pp. 479–485.

[20] Arend Rensink, Ivoka Boneva, Harmen Kastenberg, and Tom Staijen. *User Manaual for the GROOVE Tool Set*. Accessed April 10, 2015. Nov. 12, 2012. URL: http://groove.cs.utwente.nl/wp-content/uploads/usermanual1.pdf.

[21] *Thesis Project Repository*. https://bitbucket.org/ccorrodi/masters-thesis-public. Accessed April 10, 2015.

[22] Julian Tschannen, Carlo A. Furia, Martin Nordio, and Bertrand Meyer. "Usable Verification of Object-Oriented Programs by Combining Static and Dynamic Techniques". In: *Software Engineering and Formal Methods - 9th International Conference, SEFM 2011, Montevideo, Uruguay, November 14-18, 2011. Proceedings*. 2011, pp. 382–398.

[23] Scott West, Sebastian Nanz, and Bertrand Meyer. "A Modular Scheme for Deadlock Prevention in an Object-Oriented Programming Model". English. In: *Formal Methods and Software Engineering*. Ed. by JinSong Dong and Huibiao Zhu. Vol. 6447. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 597–612. ISBN: 978-3-642-16900-7.

[24] Scott West, Sebastian Nanz, and Bertrand Meyer. "Efficient and Reasonable Object-oriented Concurrency". In: *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP 2015. San Francisco, CA, USA: ACM, 2015, pp. 273–274.

[25] Eduardo Zambon and Arend Rensink. "Solving the N-Queens Problem with GROOVE - Towards a Compendium of Best Practices". In: *ECEASST* 67 (2014).