Separating computation and coordination in the design of parallel and distributed programs

Chaudron, M.R.V.

8 Related Work

An essential feature of the method of program design that is presented in this thesis is that the correctness and complexity aspects are addressed by separate models for computation and coordination. Associated with these models are formalisms for reasoning about them.

In the next section we will discuss some other programming methods and mention some commonalities and differences with the method presented in this thesis. In the subsequent section we will discuss some formalisms for reasoning about parallel systems with shared memory and compare those to the methods we have used in this thesis.

The ubiquity of coordination throughout computer science has lead to a broad field of study. Some general starting points for this area of research are [1] and [35].

8.1 Separation of Computation and Coordination

According to the method proposed in this thesis a program can be seen to consist of a computation component and a (separate) coordination component. The computation component defines *what* a program computes and is responsible for correctness. The coordination component specifies *how* a program computes and thereby defines that program's time and space complexity.

In some guise or another correctness and complexity have been recognized as the most important aspects of programs throughout the academic computer programming community. Moreover, it has long been suggested that these aspects be dealt with separately. This has lead to the approach of "program derivation by successive step-wise refinement" for the most significant programming language paradigms.

We consider the major programming paradigms and examine to which degree computation and coordination aspects are separated in associated methods of program design.

8.1.1 Functional Programming

We give a brief introduction to the ideas that underlie the functional programming paradigm. A introductory text in this area is [16].

The idea behind functional programming is that a program can be seen as a function which transforms an input into an output. In accordance with this view, the making of a functional program consists of constructing an expression that denotes a function which relates outputs to inputs in the required manner. To facilitate the definition of such expressions, functional programming languages provide a collection of primitive functions (including constants) and a method for creating new functions by giving defining equations in terms of existing functions.

A computation consists of rewriting some initial expression by successively substituting one side of a defining equation of the program by the other side, until no more rewrites are possible. The resulting term is said to be a "normal form" of the initial expression.

This rewriting process has its theoretical foundations in the λ-calculus [14]. A theoretical result from the λ-calculus states that every (well formed) expression has a unique normal form. A consequence of this result is that the order in which an expression is rewritten is irrelevant to the result. In particular, disjoint subexpressions may be rewritten in parallel.

Hence in designing a functional program, the programmer does not need to concern himself with the operational aspects of his program and may focus on the correctness of the output it yields. As a secondary concern, the programmer may want to optimize the execution of his program. To this end, the functional programming community suggested the "transformation approach":

> The transformation approach is to separate the [...] programming task in two stages. Firstly to write the program concentrating on making it as clear and understandable as possible, neglecting efficiency considerations, and then to successively transform this to more and more efficient versions using methods guaranteed to preserve the intent of the original program while improving its efficiency [41].

The methods for transforming functional programs are based on the principle of replacing equals by equals. This principle is valid for functional programs because, in contrast to imperative programs, they satisfy the principle of "referential transparency" which states that variables always denote the same quantity (within the context in

which they are defined). As a result, it is possible to use the algebraic properties of the primitives of functional languages to transform programs in an equational style. A survey of transformation techniques for functional (and logic) programs is given in [95].

The approach described in [68] suggests that this arsenal of transformation techniques can be used to increase the efficiency of Gamma programs once these are expressed in a functional programming language.

However, it is rather unsatisfactory that in order to change the behaviour of a functional program one has to change its representation (i.e. the defining equations) which is promoted for its abstraction from operational aspects. Instead of their representation, it is some execution mechanism that is external to functional programs that actually defines their behaviour. The reason for transforming programs is to increase the degree in which the representation concords with the execution style of this mechanism. In this sense, the method of functional programming has in common with that of imperative programming that programs are adjusted to match an underlying machine (albeit a more abstract one than the Von Neumann machine towards which imperative programs are tailored).

This inadequacy of functional languages to express the behaviour of programs is recognized by Darlington in [42]. There, a language akin to temporal logic is used to give a separate specification of the behavioural aspects of a program which must be satisfied by the implementation.

As alternative approach for obtaining finer control over the behaviour of functional programs, the use of annotations has been suggested by many authors. For example, Hudak's "para-functional" programming [78], Nöcker et. al.'s Concurrent Clean [92] and Cox et. al.'s language Caliban [39] propose to extend functional programming languages with annotations which include primitives for process creation and termination, combinators for sequential and parallel composition and mappings of tasks to processors. In these cases the annotations are dispersed through the program text. In contrast, we propose in this thesis to specify the behaviour textually separate from the specification of the computation. This simplifies reasoning about operational aspects without reference to the computation aspects and facilitates algebraic manipulation of the behaviour without affecting the correctness of the computation.

8.1.2 Logic Programming

Logic programming is a paradigm of computation that is based on the idea that a computation can be seen as a series of inferences in a formal logic. Logic programs consist of inference rules in a first order language (for instance predicate logic) such that their solution can be obtained by an automated-proof procedure (in particular by resolution and pattern matching). General introductions to the area of logic programming are [87] and [2].

One of the earliest accounts of an approach towards programming where the computation and complexity aspects are addressed separately is [69]. In that paper Hayes argues that a logic program can be considered to consist of a collection of deduction rules (which can be used to generate proofs) and a theorem proving mechanism which controls the application of the deduction rules.

Building on Hayes' insights, Kowalski publishes [82] in which he writes.

> [...] when *logic* is separated from *control*, it is possible to distinguish what the algorithms does, as determined by the logic component, from the manner in which it is done, as determined by the control component.

Consequently, he advocates the following approach to the design of logic programs in [84].

> The problem of developing a correct but efficient program can usually be decomposed into two simpler subproblems:
>
> 1. Specification. The first task is to specify the problem to be solved and the information which is needed for its solution.
> 2. Efficiency. Inefficiencies implicit in the problem specification can then be identified and removed, transforming the specification into an effective program.

The phases of this method are supported by the following techniques

- A method for massaging a specification in predicate logic into the format required by logic programming languages (see Chapter 10 of [83]).
- A collection of correctness-preserving program transformations for increasing the efficiency of the program (based on the seminal work [20]; a recent overview is given in [95]).

These articles by Hayes and Kowalski suggest that the ideas for separately developing a program's computation and coordination aspects were present at an early stage of the evolution of logic programming. However, instead of putting this idea to use, research continued towards the design of autonomous proof procedures which would be sufficiently general to ensure satisfactory performance for all possible logic programs. This focus of attention was justified as follows

> The control component can be expressed by the programmer in a separate control language; or it can be determined by the program executor itself. The provision of a separate control language allows the programmer to advise the problem-solver about program execution and is suitable for the more experienced programmer. The determination of control by the program executor, on the other hand, relieves the programmer of the need to specify control altogether and is more useful for the inexperienced programmer, the casual database user, and even the expert programmer during the early stages of program development. (from p. 127 of [83])

Because logic programming languages were not designed for describing operational aspects, they were extended for parallel programming by adding explicit (extra-logical) mechanisms for synchronization and communication. Consequently, the computation and coordination aspects of (parallel) logic programs are intertwined in a single program text. Overviews of approaches towards parallel logic programming are given in [109] and [37]. As a result, there is no general method for reasoning solely about the behavioural aspects of logic programs. This becomes particularly desirable for programs which deviate from the default behaviour through the use of extra-logical features such as the cut operator or synchronization and communication primitives.

A relation between the Gamma model and logic programming is described in [33]. There, the Gammalög programming language is presented which shares features of both logic programming and the Gamma model. A sequential prototype of this language has been implemented using the programming language Gödel [71].

8.1.3 Imperative Programming

The functional and logic programming paradigms are based on the idea that a program is an abstract object that can be captured by specifying its mathematical properties from a particular perspective. In contrast, the imperative programming paradigm has

evolved as a method for instructing a machine to behave in a certain manner. Data is stored in variables which essentially are named addresses in the memory of a computer. The basic unit of computation is the assignment statement which stores a value in a variable. Programs are built by defining an order of executing assignments. To this end, the following control-flow constructs are typically employed: sequential compositions, conditional composition (if-then-else) and the iterative constructs (for, while and repeat).

The following quotation (from p. 237 of [61]) illustrates that in the imperative programming community the benefits of structuring the design of a program in phases is also recognized.

> The programmer has two main concerns: correctness and efficiency [...] When faced with any large task, it is usually best to put aside some of its aspects and concentrate on the others [...]. This important principle is called *separation of concerns*

The methodology for the design of sequential programs differs somewhat from those of the functional and logical programming paradigms because due to their lack of useful mathematical properties, imperative programs do not lend themselves well for transformation. However, a transformational method of program development can be realized if a formalism is used which is able to express both specifications and programs. We will discuss two of such formalisms which have also been put forward as methods for the design of parallel programs.

Action Systems

The action system formalism for parallel computing was introduced in [7, 8]. An action system consists of a set of guarded assignment statements that co-operate through shared variables. The statements of an action system may be executed in an arbitrary order; statements may be executed in parallel as long as the actions do not have any variables in common. Actions are executed atomically; i.e. without interference from any other action in the system.

Initially, a calculus of refinement of action systems was based on weakest preconditions [4, 5]. Based on this notion of refinement a step-wise method for the development of parallel programs is presented in [6, 9, 108]. Later, methods of refinement with a larger emphasis on the refinement of behavioural aspects of action systems were described in [113] and [10]. There, refinement is defined in terms of traces of events of action systems.

The action systems model has in common with Gamma that it consists of a collection of actions that is executed in an arbitrary order. However, the notions of refinement for action systems take sequential behaviour as the default and gradually introduce opportunities for parallel execution.

> [..] the goal is to transform a more or less sequential algorithm or algorithm specification into an action system that can be executed in a highly parallel fashion, so we need special refinement rules to introduce parallelism into the execution. (p. 122 [9]).

This is dual to the method of refinement proposed in this thesis which starts with parallel behaviour and gradually increases sequentiality.

Furthermore, the kind of refinement employed by action systems also differs from the one used in this thesis. Refinement of action systems is based on the idea of "action refinement"; i.e. an action gets replaced by a collection of actions that achieve (a refinement of) the same relation between input and output. When using this method of refinement, a program is developed from a specification by successively inventing the actions that can be used to implement a specification. The method of action refinement implies that both computation and coordination issues are resolved in a single refinement step.

Notwithstanding these differences, there are some efforts in the action systems community which tend toward the separate development of computation and coordination aspects.

For instance, in [8] a formal method for decentralizing the control of a given action system is described. Decentralization effectively consists of decomposing a single action system into a layered system where a higher layer controls the order of computations of lower layers (without interfering with the computations). On a conceptual level, this is related to the coordination approach.

A benefit of this approach is that it straightforwardly caters for multiple layers of coordination which can be thought of as higher-order coordination. A draw-back of this approach is that the coordination is encoded by means of control-variables which are used to synchronize the actions and transport values back and forth between components of an action system. This method is not designed for the specification of coordination strategies and, according to [108], results in tedious verification steps.

The details of refinements can be suppressed by using a collection of transformation laws. Such a collection of laws for the transformation of sequential action systems into

parallel action systems is described by Sere in [108].

In [70], Hedman, Kok and Sere propose a structured method for the refinement of action systems. This approach is based on specifying action systems as the (prioritized) composition of a computation part and a coordination part. Following this discipline, the computation and coordination parts can, to a large extent, be refined independently. However, because coordination is achieved through shared variables, a data-refinement in either the computation part or the coordination part requires that additional conditions be checked in both parts with respect to these variables.

Unity

The UNITY framework, introduced by Chandy and Misra in [23], consists of a programming language and a programming logic. A UNITY program consists of a set of guarded (multiple) assignment statements. The execution mechanism resembles that of action systems and Gamma in that this set of statements is repeatedly executed in a nondeterministic order until a fixed-point is reached.

The programming logic is based on a small set of temporal properties (of sequences of states). UNITY's approach to program development is to start with a specification (in predicate logic) and successively introduce more concrete actions by action refinement. A typical example of such a refinement is that a (assignment) statement is replaced by a collection of statements that operate upon a more detailed representation of the data but satisfy the same temporal properties. The relation of this approach to the approach followed in this thesis is essentially the same as that described for action systems.

A methodological difference between the UNITY and action systems approaches is that UNITY is aimed at refining specifications, while action systems are aimed at refining programs. This distinction has become less strict by the extension of the UNITY method with the structuring mechanism of procedures and local variables in [111].

The nondeterministic execution mechanism of UNITY makes it in principle susceptible to the superposition of a coordination component. However, such an approach is not taken (nor could it be found in the current literature) and the ordering of execution of UNITY programs is defined by operators for sequential composition, (may) parallel composition and synchronous (must) parallel composition.

This obstructs the mutually separate refinement of the computations and coordination and prevents the re-use of coordination components for computation components with similar structure.

Concluding Remarks

The approaches of both action systems and UNITY introduce an ordering on the computation by incorporating coordination structures in the same program text. This means that programs have to be redesigned for different architectures.

Furthermore, the use of the imperative style of representing data by variables (rather than tuples) imposes premature constraints on the execution. As a consequence, special attention has to be paid to the discovery of potential parallelism during the process of refinement.

8.2 Reasoning about Parallel Shared Memory Programs

A large portion of this thesis is devoted to the development of formal methods for reasoning about the correctness and refinement of parallel programs which share a common memory. In this section we survey some other formal methods that have been proposed for reasoning about the correctness of such programs. In particular we examine in what way they support the development of programs by step-wise refinement.

We classify the methods according to the style of semantics that is used. We do not claim that these categories are disjunct - some methods may fit in more than one category.

8.2.1 Axiomatic/Assertional Reasoning

In [73] Hoare suggested the use of assertions to reason about (partial) correctness of sequential programs. This method uses triples $P\ \{S\}\ Q$, where P and Q are predicates and S is a statement, to mean that, if S is executed when pre-condition P holds, then post-condition Q holds if S terminates.

Starting from an axiom that defines the effect of an assignment, the effect of a program can be derived from its components using a set of inference rules for all combinators in the programming language.

Extensions of this approach for dealing with primitives for coordinating concurrent execution or mutual exclusion of parallel programs have been proposed; e.g. in [75], [93], [85]. These extensions have in common that they are based on a formal notion of non-interference. Effectively, these methods require that the proof of a program be

formulated in terms of properties that are interference-free. These are essentially the same kind of properties that are used for proving convex refinements.

The action systems approach is also essentially an assertional method. This approach is closely related to the methods of imperative program design that we discussed in Section 8.1.3.

An axiomatic method for reasoning about Gamma programs is presented in [52] and [51]. This is based on the axiomatization of the pre- and post condition of the rewrite rules of Gamma programs. In composing these conditions, this approach takes interference into account by allowing the precondition of a transition to differ from the postcondition of the preceding transition. This corresponds to the way in which interference is modelled in the framework of generic refinement. This approach yields a compositional method for reasoning about input-output refinement of Gamma programs. However, the method tends to be impractical for manual proofs and provides little support for the design of programs by step-wise refinement.

A systematic approach to the construction of an axiomatic-style program logic for Gamma is pursued in [57]. The programme of that paper is to derive a program logic by applying Abramsky's domain theory in logical form to a (denotational) resumption semantics for Gamma.

8.2.2 Denotational Methods

The aim of the denotational semantics is to find ways of defining the meaning of programs in a compositional manner; i.e. methods for calculating the meaning of a program from the meanings of its constituent parts.

In the denotational style, several methods based on the concept of transition traces have been proposed for assigning meanings to programs that communicate asynchronously via shared memory. We will discuss some of these methods in this section.

For an imperative parallel language with shared memory, Brookes presents in [19] a number of laws, based on transition traces, which resemble some of the stateless laws that we presented in Section 4.4.2. These laws have in common that they are independent from the current state. This correspondence suggests that Brookes' law $C_1; (C_2 \parallel C_3) \sqsubseteq (C_1; C_2) \parallel C_3$ can be extended to the analogue of the distributive law proven by Lemma 4.4.13.

The validation of refinement methods in this thesis suggest that laws that do not exploit properties of the state are not sufficiently powerful to justify some useful refine-

ments that one would reasonably expect to hold. Our generic framework for refinement suggests that this insufficiency is due to the fact that transition traces (as well as stateless simulation) allow arbitrary interference to occur during a computation. It is to be expected that the approach based on transition traces can be used to derive more powerful laws by limiting the possible interferences. One attempt at limiting the interference in a transition trace based method is described by Dingel in [48] where he investigates how the interferences allowed by transition traces can be linked to the context of a program. In this thesis, the idea of relating the interference of a program to its context led to the development of convex refinement.

It would be interesting to develop a theory of transition traces which is parameterized by the interference. This could yield a generic framework, analogous to the one based on simulation described in Chapter 5, which allows a more flexible way of formalizing assumptions about the possible interferences.

The general applicability of the transition traces approach to concurrent languages that communicate through asynchronous communication is shown by De Boer et al. in [18]. There it is compared with the failure semantics which is used for assigning meaning to languages based on synchronous communication.

A transition traces approach for Gamma has been developed by Sands in [105].

An interesting feature that is laid bare by the simulation approach is the distinction between strong and weak refinements. By their construction, transition traces are closed under "stuttering". As a result they seem insensitive to this difference.

An interesting alternative for using a multi-step transition system for modelling the concurrent execution of rewrite rule is the pomset model [100]. We chose to base our method of refinement on bisimulation because it has a powerful proof method associated with it.

8.2.3 Temporal Logic

Temporal logics (e.g. [97], [86]) use properties of sequences of states or sequences of actions for specifying and reasoning about parallel systems.

Because of its non-operational style of reasoning, a temporal logic is used in the UNITY [23] approach. programs based on temporal logic is illustrated by Singh in [110]. There Singh presents a refinement rule which justifies the strengthening of guards of UNITY statements. This rule closely resembles the convex strengthening law that we prove for our coordination language by Lemma 6.2.1.

The applicability of temporal logic to Gamma is illustrated by Reynolds in [103] where he uses it for defining a semantics for Gamma. It would be interesting to extend this approach to schedules and investigate whether it could serve as the basis of a method of refinement of schedules. Since temporal logics are well suited for dealing with reactive processes, such a method could complement the more output-oriented approach followed in this thesis.

8.2.4 Algebraic Methods

There is a large body of work on algebraic methods for the description of communicating processes (most notably ACP [11], CCS [90], CSP [76]). The algebraic approaches towards reasoning about parallel systems consists of using variables (and constants) to denote events and using operators, such as sequential and parallel composition, as combinators for processes. According to this approach, properties of processes can be described in an algebraic framework.

In this area of research, the main focus is on processes that have private state (and communicate through message passing). In order to obtain a decoupling between computation and communication we used a programming model which employs asynchronous communication via a shared data-space (i.e. shared state). As a consequence of this difference, the behaviour of scheduled Gamma programs is defined in terms of configurations $\langle s, M\rangle$ which consist of a schedule component (a process part) and a multiset component (a state part).

This difference from the private-state approaches prohibits that (bi)simulation-based notions of equivalence yield a (pre)congruence relation over schedules. Hence, it complicates the construction of a method for algebraic reasoning about shared-memory processes. However, in Chapter 5 we have shown how (pre)congruent refinement (equivalence) relations can be obtained. We will discuss some alternative approaches for solving this problem that have been proposed in the literature.

In [63] Groote and Ponse first consider a language which contains combinators for prefixing, nondeterministic choice and guards. They employ a two-stage construction to obtain a congruence for this language. Firstly, an equivalence relation over process-state configurations is defined using bisimulation . Next, an equivalence over programs is defined which equates two processes if they are bisimilar for all possible initial states. The generic framework of refinement from Chapter 5 suggests that the quantification of the state-component in the notion of equivalence can be interpreted as taking all possible

interference into account. From this point of view, the approach of Groote and Ponse defines a notion which allows interference only before the first action, and no interference from the the first action onwards. This approach seems prompted by the fact that it technically (from a mathematical point of view) solves the problem, however it does not reflect a sensible assumption about interference in a shared memory setting.

Next, the language considered in [63] is extended with operators for parallel composition. The notion of equality of processes sketched above does not yield a precongruence in this setting. This is solved by adapting the notion of bisimulation such that it allows interference at every stage of execution.

This latter approach is essentially the same as that followed in [34] where Ciancarini, Gorrieri & Zavattaro develop a bisimulation-based equivalence for Gamma programs (based on a Plotkin-style operational semantics [96]) by universally quantifying over the multisets.

The universal quantification of the state-component of these notions of bisimulation corresponds to the type of quantification used to define stateless refinement. Hence, these notions do not support refinements that depend on properties of the state. Based on our experience with notions of refinement in this thesis, we expect that these stateless-like notions are not (by themselves) sufficiently powerful to justify all refinements that one would reasonably expect to hold (and hence expect to be able to use) in a shared memory setting.

An alternative approach is obtained by avoiding the distinction between operations and data by considering both of these as elements of the same algebra. This essentially boils down to considering a datum as a process that can offer its value and may engage in a communication with processes that are looking to use this datum. This idea has been used by Ciancarini et al. in [36] and by De Nicola and Pugliese in [44, 45] for giving semantics for Linda (see [22]).

These approaches manipulate a mathematical structure which includes both the data structure and control structure. In our opinion, this obfuscates the coherence of the coordination structure. As a result it is not clear how the coordination structure can be refined in a modular way. Also, it is unclear how properties of the data can be used in refinements.

Besides the best established algebraic approaches, a noteworthy effort is that of the "Calculus of Broadcasting Systems" [98, 99] - CBS for short. CBS is a CCS-like calculus which has broadcasting rather than hand-shaking as basic primitive for communication. In [107] a translation of a class of Gamma programs into CBS is examined.

As a consequence of choosing broadcasting as communication primitive, there are a number of similarities with the programming model we used in this thesis. In CBS holds that once a datum has been broadcast, it is available to all "listeners". Similarly, we have in Gamma that once a datum has been inserted in the multiset it is available to all rewrite rules that wish to use it. Related to this is the fact that a listener in CBS and a rewrite rule in Gamma do not have to identify the source (broadcasting process or creating rewrite rule) of a datum in order to use it. Hence, analogous to the Gamma model, the broadcast communication abstracts from locality.

However, in contrast to the shared state of the Gamma model, CBS has processes which have private state. A further difference is that it is inherent in the principle of broadcasting that only one single process may broadcast at a time. Thus only one datum is available for all actions that are looking to engage in a communication. This limits the number of computations that can be performed at any time in a way that is not inherent to the logic of the problem.

In the setting of process algebras with private state and message-passing there have been some efforts towards a generic theory of equivalence aimed at obtaining precongruences over process-terms [64], [17] and [46]. Just as our approach, these are based on the idea of using (bi)simulation as notion of refinement (equivalence) over behaviours. In contrast to our approach, these efforts have fixed the notion of equivalence (as bisimulation) and predict that it is a (pre)congruence if the semantic rules that define the structural operational semantics fit certain formats. Therefore, these approaches are more oriented towards investigating the design of (primitives for) programming languages, rather than investigating a generic framework of notions of refinement.

8.3 Concluding Remarks

> *[...] one of the outstanding challenges in concurrency is to find the right marriage between logical and behavioural approaches.*
> Robin Milner, p. 4, [90]

During the stages of our method for program design, we separately focus on computation (functionality) and coordination (behaviour). These phases lend themselves for different methods of reasoning.

We can adequately reason about the correctness of a computation by means of a logic which abstracts from behaviour. Subsequently, during the design of a coordination

strategy emphasis is on the behavioural aspects of a system and this is reflected by the methods which we have proposed in support of that stage.

It is at the behavioural level that the additional complexity of reasoning about parallel systems manifests itself. This complexity stems from the interaction of behaviour of individual components. All of the methods for reasoning about parallel systems deal with interaction between individual components in one way or another. For instance by excluding interference in time (critical sections) or space (by ensuring that control over a variables is local). In our framework interaction plays a central rôle at the level of behaviour by using the degree of interference as a parameter in our theory of refinement.