Scheduled protocol programming

Dokter, K.P.C.

Chapter 9

Conclusion

9.1 Summary

With the advent of multicore processors [ABD+09] and data centers [Kan09] computer hardware has become increasingly parallel, which allows one to run multiple pieces of software at the same time on different machines. Programmers can enjoy the fruits of parallelism by partitioning software into relatively independent components and running these components simultaneously.

Of course, these components are not completely independent: they must cooperate to achieve the common goal of the complete software system. Such cooperation of components requires shared resources and coordination. As explained in Part I, coordination is best expressed in an explicit interaction protocol that clearly defines the interactions amongst all components in the software. An explicit interaction protocol not only improves code structure, but also enables automated analysis of the protocol [Klü12], and improved execution efficiency by compiler optimizations [Jon16]. In Chapter 2, we compare two coordination languages, BIP and Reo, and offer formal translations between them. Our comparison reveals differences between BIP and Reo with respect to composition and priority. The difference in composition leads to the proposal of a composition operator for data-sensitive BIP architectures.

We aim to narrow the 'priority gap' between BIP and Reo in Chapter 3 by expanding the theory of soft constraint automata with memory cells and bipolar preference values. These soft constraint automata offer a semantics for Reo wherein bipolar preference values can express a weak form of priority, such as context-sensitivity. Our approach to context-sensitivity differs significantly from other approaches (that do not model preferences as explicit values), such as connector coloring [CCA07], dual ports [JKA11], intentional automata [CNR11], augmented Büchi automata of records [IBC08], and guarded automata [BCS12]. Although we consider context-sensitivity in the realm of Reo, we stress that context-sensitivity is a fundamental concept that applies to languages other than Reo.

Part II of our thesis focuses on the generation of executable code for a given interaction protocol that merges its coordinated components into a single concurrent software. We temporarily ignore the coordinated components and study the

interaction protocol in isolation. The basis of any compiler is a well-defined input language. Many Reo tools use a plugin in the Eclipse editor as a graphical editor for Reo connectors. The graphical editor not only commits the user to the Eclipse editor, but also lacks important features such as parameter passing, iteration, recursion, or conditional construction of connectors. Some Reo tools therefore develop their own textual language, each having a specific Reo semantics in mind. In Chapter 4, we propose a textual language, called Treo, that does not commit to any specific semantics.

We use the freedom of the semantics in Treo to propose a new semantics that represents Reo connectors as temporal logic formulas called stream constraints. Such constraints can be written in may equivalent forms. Since composition is conjunction, the composite constraint grows linearly in size. However, compilation of such a conjunction is hard and requires a constrain solver (possibly at run time). Chapter 5 identifies the rule-based form as a form that balances the trade-off between composition and compilation of constraints. A rule in a rule-based form corresponds naturally to a loosely coupled thread or process, which makes compilation straightforward. The composition of rule-based constraints grows linearly most many practical cases (such as the $\mathsf{Alternator}_k$ in Figure 5.3), and grows exponentially only in the worst case. We implemented a Reo compiler based on stream constraints, and showed that it outperforms the state-of-the-art Reo compiler ([JKA17]).

Reflecting on our stream constraints, our main observation is the discrepancy with respect to concurrency between Reo connectors and their formal semantics [JA12]. While Reo connectors are considered concurrent, almost all semantics for Reo are expressed in an inherently sequential model (such as streams or automata). The encoding of Reo connectors into zero-safe (Petri) nets is the only exception. This mismatch results in ad hoc extraction of concurrency via techniques like 'synchronous region decomposition' [JCP16], 'local multiplication' [Jon16], or our rule-based form in Chapter 5. In Chapter 6, we propose an inherently concurrent semantics for Reo connectors as Multilabeled Petri nets, which turns the ad hoc extraction of concurrency from our rule-based form into the standard interpretation of concurrency in Petri nets. While the syntax and semantics of Petri nets is standard, our main contribution is a composition operator on Petri nets that mimics the composition in stream constraints. As a result, we obtain a simple expressive Reo semantics that can be effectively compiled into efficient code.

In Part III, we study the effect of the interaction protocol schedulability of the complete software (including the coordinated components). In order to compute high quality schedules for a software system, we must know how much processing time (or work) each component requires. To this end, we develop work automata in Chapter 7, which express, for a fixed number of components, how much work each component can/must do. We develop a gluing technique that allows us to minimize the state space of a work automaton to potentially a single state with a complex invariant. This invariant exposes mutual exclusion as holes in a higher-dimensional space. If a scheduler can avoid such holes, we can potentially drop the (costly) locks that prevent the application from transgressing into these holes.

Next, we use work automata to develop a game theoretic scheduling framework. In general, the scheduler decides which components can run, while the application decides how to resolve non-determinism during execution. In Chapter 8 we for-

malize this game as a two-player zero-sum game played on a graph, and slightly adapt existing solutions to compute a scheduling strategy for a small cyclo-static dataflow application that optimizes throughput.

The most straightforward solution to implement our synthesized strategy is to replace the default operating system scheduler with a custom application-specific scheduler. While this approach is possible, it is a non-trivial, costly task that requires administrative rights on the operating system. We present a novel alternative that implements the synthesized strategy in the software, while keeping the default operating system scheduler in place. First, we transform the resulting strategy into a scheduling protocol, and subsequently compose this scheduling protocol with the original protocol of the application. As a result, a general-purpose operating system scheduler (which schedules all non-blocked components in a round-robin fashion) will closely follow our optimal strategy. As a result, we avoid complex custom schedulers, while still obtaining our scheduling goals.

9.2 Future work

In this thesis, we demonstrated the practical use of the schedule-protocol duality from Chapter 1 by applying it to the special case of cyclo-static dataflow software. However, the applications of the schedule-protocol duality are not limited to this special case, and further research can study the implications of this duality on cyber-physical software and real-time systems. These more general systems often have real-time constraints and alternative objectives (other than throughput), which we do not consider in the current thesis.

Even if a program is represented as a small Petri net, its scheduling game can have a very large number of positions. The scheduling game of the cyclo-static dataflow program from Chapter 8 turned out to be just small enough to be solved on our personal laptop, but solving the scheduling game for larger programs requires superior hardware like a computer cluster. To handle even larger applications, we can improve the scalability of schedule synthesis by using different solvers that avoid the state-space explosion. One direction is to search for classes of programs that can be scheduled by existing powerful solvers (like ILP solvers, SAT solvers, or SMT solvers). Another direction is to develop scheduling heuristics that simplify scheduling synthesis at the cost of suboptimal solutions.

The work automata introduced in Chapter 7 encode all relevant scheduling information and we use them as input to our scheduling framework. In this thesis, we assumed that these work automata are given and put our focus on the resulting scheduling problem. Although the problem of finding the work automata of a given program is non-trivial, the search can be automated, if all necessary details (such as the exact machine instructions and the hardware architecture) are known. Even if not all necessary details are known, we can still approximate the size of the workloads through experimentation. The workload can then be represented as a distribution (rather than a single value), which can be encoded as a work automaton via non-deterministic transitions.