CSP as a Coordination Language

Moritz Kleine

CSP as a Coordination Language

Moritz Kleine

Technische Universität Berlin
Institute for Software Engineering and Theoretical Computer Science
Berlin, Germany
mkleine@cs.tu-berlin.de

Abstract. Coordination languages allow us to separate interaction behavior from the sequential functional aspects of the components of concurrent systems. This helps us to reduce the complexities of such systems making them easier to design and to understand. However, there is still a gap between formal approaches to coordination and their implementation in programming languages. For example, CSP is often used as a coordination model but only subsets of CSP are supported by programming languages (e. g., occam) or frameworks (e. g., JCSP). In this paper, we present our approach to using a more complete CSP as a coordination language. Our approach allows us to use standard CSP tools for verifying the coordination processes of a system and to use these processes at runtime to coordinate the systems' components.

1 Introduction

A major problem in developing concurrent software is that the additional complexity that comes with concurrency often causes unexpected asynchronous behavior that quite often manifests in subtle bugs. Formal methods such as Communicating Sequential Processes (CSP) and formal designs help to avoid such bugs during the design phase of a system. However, finding methods for deriving implementations from CSP-based models is an active field of research. It is, for example, not obvious how to integrate CSP with internal actions of a system, because CSP abstracts from internal actions to a great extent. Coordination languages such as Linda [2] offer an approach to taming the complexities of concurrent programs. This family of languages targets the description of the interaction behavior of a system's components separating the interactions from the sequential functional properties of the components. While coordination languages target the implementation level description of concurrent or distributed systems, process algebras target their design and are more tailored to (mechanically) verify systems.

In this paper, we present our approach to using CSP as a coordination language. The approach is to coordinate non-atomic actions of a concurrent system by a CSP-based coordination environment in a noninvasive way. We present a model of a coordination environment that interprets a process simulating its truly concurrent semantics (where concurrent events may be performed at the

same time and not only interleaved) within the standard interleaving operational semantics of CSP. The abstractions inherent to the process algebra are undone by regarding an event as a terminating (not necessarily atomic) action and a hidden transition as an internal action of the final system. Whenever the coordination environment performs a (possibly hidden) event, it performs the action mapped to that particular event. Actions can implement arbitrary computations, even communication, including shared-memory communication and message-passing. Our approach allows us to use CSP for the design and implementation of concurrent systems by adding a coordination environment to a (sequential) host language. The use of CSP enables us to perform deadlock-checking and more advanced properties (by means of refinement and LTL model checking) at design time.

In the next section, CSP is introduced and approaches using CSP for the engineering of coordinated concurrent systems are presented. Abstractions built into the language and their relevance for coordination are discussed in Section 3. The model of a coordination environment is presented in Section 4. Section 5 presents a proof obligation to ensure data independence of concurrent actions. Important properties of coordination processes are characterized and discussed in Section 6. The Java implementation supporting our approach is presented in Section 7. In that section, we also give pointers to further work. We then present related work in Section 8. The conclusions of this paper are given in Section 9.

2 CSP and Coordination

The process algebra Communicating Sequential Processes (CSP) [6,14], provides an algebraic notation based on *events*, process names and process operators tailored for the concise modeling of current systems. Concurrent systems are modeled as *processes* that perform events. If a process offers an event with which its environment agrees to synchronize, the event is performed. Events are both atomic and instantaneous.

Events are commonly regarded as rendezvous communication between processes. From this viewpoint, processes are anonymous entities communicating synchronously over named channels. Accordingly, an event models the occurrence of a communication identified by the channel name and the message being sent over it. Messages can only be sent if the receiver is willing to accept them. Events can also be regarded as abstractions of atomic actions of a system in the understanding that duration of actions can be modeled by splitting events into start and end events.

CSP is equipped with operational, denotational and algebraic semantics. The concept of refinement facilitates the step-wise development of processes by gradually restricting their behaviors. In this context, CSP enjoys the property of *compositionality*: given a process that satisfies some specification and another process refining a part of the first process, we may replace that part with the second process and obtain a new process that also satisfies the specification. Modeling, exploration and verification of processes is supported by a number of

industrial-strength tools. The automatic refinement checker FDR [14], for example, proves or refutes refinement assertions in the denotational models *traces*, *stable failures* and *failures-divergences*.

In [15], Schneider and Treharne present a combination of CSP and B [1] that is now known as CSP‖B. The idea of CSP‖B is to separate the specification of a component into state related and interaction properties. The B method is used to express requirements on state of the components and their coordination is expressed in CSP. CSP events are associated with operations, hence assuming atomicity of operations.

CSP++ [5], is a framework realizing CSP concurrency on top of POSIX threads for C++. The framework defines a whole development life-cycle starting with a CSP_M specification being refined down to a CSP implementation model. The implementation model is finally translated into C++ using the CSP++ framework. This framework implements channels as an inter-process communication primitive and allows us to bind user-coded functions to events that are executed when the event is performed. The CSP++ framework implements the occam-style CSP supporting sequential composition, external choice, interleaving and parallel composition.

JCSP [18] is a well-known Java library offering CSP concepts as a foundation for developing concurrent systems in an event-based style. In this framework, processes communicate over channels which are basically buffers. JCSP realizes CSP's synchronous communication between Java threads by blocking the send operation until the value is read by its receiver. This is realized using the Java primitives *synchronize*, *wait* and *notify*. Processes (implemented as Java threads) are not allowed to invoke each other's methods but they may be combined to wait passively on a number of alternative events. The external generation of such an event triggers the processes into action according to the CSP semantics of the process operator combining the events.

3 Unravelling Abstractions

This section explains, how the abstractions built into CSP can be unravelled to coordinate concurrent systems. The arguments presented here are based on the following assumptions.

1. CSP offers a rich set of operators to facilitate the concise modeling of concurrent systems. This set is intendedly non-minimal and offers different operators to describe semantically equivalent processes. The idea is that the operators model abstractions of different implementations. The coordination environment must be able to unravel these abstractions accordingly.
2. Once a process is proved to satisfy the interaction behavior of the components of a system, the process should be directly usable as a coordination process without being refined further (even if the process is a nondeterministic one).
3. Coordination processes still abstract from data being used in the implementation.

4. Although the standard CSP semantics are interleaving ones, a coordination environment must be able to profit from true concurrency if that is offered by the underlying computing hardware.
5. Coordination processes define the external and internal interactions of a system.

3.1 Timeout, Hiding and Nondeterminism

The operators timeout, hiding and internal choice, offer abstractions that allow us to develop concise models of concurrent systems but which have to be undone for the implementation of a conforming system For example, although

$$P \triangleright Q = (P \,\Box\, Q) \sqcap Q$$

holds, the timeout operator offers a convenient abstraction of a process that switches automatically to Q if P fails to perform a visible event within a certain time interval. From an implementation point of view this is fundamentally different from $(P \,\Box\, Q) \sqcap Q$ which can be understood as a process deciding to offer either the initials of P and Q for synchronization or just the initials of Q. Thus, our approach unravels the abstractions built into the timeout operator differently than that built into its semantically equivalent version. On the implementation level, $P \triangleright Q$ is implemented as a timeout (e. g., using a timer), while $(P \,\Box\, Q) \sqcap Q$ models a combination of internal and external influences (independent of time).

In the semantic framework of CSP, timeout is introduced by hiding. As explained above, our understanding of a timeout is somewhat different. Analogously, we deal with hiding different from its treatment on the semantic level of CSP. There, hiding abstracts events such that the following equality holds:

$$(a \rightarrow STOP) \setminus \{a\} = STOP\,.$$

Nevertheless, in the operational semantics $(a \rightarrow STOP) \setminus \{a\}$ performs a hidden action before evolving to STOP. Our approach adopts this understanding and takes hidden actions into account. A coordination process like $P \setminus A$ removes A from the externally visible events and specifies that the system cannot synchronize with its environment on any event from A. However, within P, the events in A are available for synchronization, of course.

A similar argument applies to

$$a \rightarrow P \,\Box\, a \rightarrow Q = a \rightarrow (P \sqcap Q)\,.$$

The left-hand side models a system that offers two similar actions for synchronization but its future depends on the external decision which one of the two exclusive actions is chosen. The right-hand side models a system offering a single action to its environment and then decides internally if it continues as P or as Q. Accordingly, $P \sqcap Q$ is understood as a system performing an internal action

to make the decision between P and Q. Thus, when used in a coordination process, the internal (nondeterministic) choice gives rise to a single internal action whose outcome determines which one of the processes is chosen. There are more reasons for allowing the internal choice to be used in a coordination process. For example, it is quite often the case that nondeterministic models of a system are sufficient to express certain properties (e. g., deadlock-freedom). Once a process is proved to express the interaction requirements of a system it is clearly desirable to take it as the coordination process and to implement the missing details on the level of some programming language instead of further refining the CSP model.

3.2 Duration, Conflict and Concurrency

Events are assumed to be instantaneous and atomic. In [6], Hoare proposes to unravel this abstraction by splitting an event into start and end event to model duration of the original event. Our approach adopts this idea. Events are split into start and end events and the action that realizes the original event is performed between these two. It is important to note that actions can be of arbitrary granularity (i.e. just a few basic operations of a processing unit or a long running service) and may even be internally concurrent. The single requirement is that it is guaranteed to terminate eventually.

Furthermore, we believe that concurrency must be distinguished from choice for the purposes of a coordination language. Designers of concurrent systems should be able to specify which parts of a system may be executed simultaneously (truly concurrent) and which are mutual exclusive. For example, the processes

$$P = a \rightarrow b \rightarrow STOP \;\Box\; b \rightarrow a \rightarrow STOP$$
$$Q = a \rightarrow STOP \;|_{\emptyset}|\; b \rightarrow STOP$$

are equivalent in the standard CSP models (e. g., traces, stable failures and failures-divergences) but we think of them as describing different systems. P describes a system that must perform a and b exclusively while Q may perform a and b at the same time. This observation is of theoretical and practical importance for a CSP-based approach to coordination. The theoretical issue is that the standard semantics for CSP are interleaving ones and thus do not distinguish choice from concurrency. The practical issue is that a concurrent program should be able to profit from the gains promised by concurrency instead of being limited to purely sequential runs (due to interleaving).

There are two ways of understanding a CSP event. Either as a hand-shaken communication of parallel processes (or components) or as an abstraction of some sequential action (not necessarily related to communication). The former understanding gives rise to implementations of channels that can be used by software developers to build their programs using CSP-style communication. Operators like hiding or internal choice are not supported by these approaches. As explained above, our approach interprets events as arbitrary actions that are

performed between the start and end events of a split event in the coordination process.

Interestingly, as shown in [8], the splitting of events allows us to distinguish choice from concurrency. In that paper, a syntactical transformation T of processes is presented that splits the events in its argument (even the hidden ones) of its input process into start and end events and relabels them in such a way that hidden transitions become observable again. T allows us to determine pairs of possibly concurrent events taking hidden events into account. T distinguishes *concurrency* from *conflict* in that start and end events of concurrent events may interleave while conflict relates to exclusive start evens. The transitions $x, y \in \Sigma \cup \{\checkmark, \tau\}$, leading from P to the states P_x, P_y, $P_x \neq P_y$ respectively, are conflicting in P if x cannot fire in P_y or y cannot fire in P_x. This deliberately includes the case $x = y$, because events can be auto-conflicting (if they lead to different states). This also extends to τ transitions, because conflicts between those are also possible (as, e. g., in $(a \rightarrow P \,\Box\, b \rightarrow Q) \setminus \{a, b\}$). Furthermore, T allows us to simulate truly concurrent CSP within the framework of standard interleaving CSP. This is beneficial because these semantics are well-developed, well-documented and supported by a number of industrial-strength tools such as FDR and ProB.

Any of the choices presented in this section is justified by the standard semantics of (untimed) CSP. An important feature is that nondeterminism is supported on the design level but resolved on the implementation level. The advantage is simplicity of reasoning about the design (e. g., deadlock-freedom is often provable on a quite abstract design level).

4 Designing a Coordination Environment

The ideas presented in the previous section are supported by the transformation T presented in [8]. The transformation splits events into start and end events, introduces fresh hidden actions for internal choices and timeouts, and takes internal transitions into account. T gives rise to a coordination environment that simulates the truly concurrent version of a CSP coordination process at runtime and starts a user-defined function (UDF) after performing the start event of a split event and performs the respective end event after termination of the UDF. A UDF is the implementation of a terminating action provided by the user.

The intent of the coordination environment is to enable coordination of a concurrent system in a *noninvasive* way separating *interaction* and *data* independently from a specific target language.

The notion of noninvasiveness means that existing implementations of components do not need to be modified to be coordinated. The coordination is done on top of the component implementations provided by the user encapsulating the coordinated components, taking their operations as UDFs implementing the actions of the final system.

The separation of interaction and data is important with respect to the modeling of the coordination process (which may involve data but which does not necessarily relate to data being communicated or computed by the actions of the system) and with respect to the final combination of the coordination process with the UDFs of the system. The first aspect means that the variables used in a coordination process are independent of the actual values being communicated and processed by the system (unlike in CSP‖B [15], for example). The second aspect means that the UDFs being mapped to possibly concurrent events may not modify data being shared amongst themselves.

The design-flow associated with our approach allows the concurrency structure and the UDFs to be developed and verified independently up to the point when the system is assembled to a concrete executable concurrent system (by combining the coordination process with the UDFs implementing the actions). Only then sets of possibly concurrent events must be identified and it must be proved that the UDFs do not introduce data races. This issue is discussed in Section 5. The following subsections present our understanding of how a coordination environment should perform actions and enable actions to be chosen internally or by the environment.

4.1 Performing actions

The purpose of the coordination environment is not only to coordinate concurrent parts of a system but also to execute UDFs when performing actions and to assign the execution of UDFs to threads. This is done while performing an event or a τ-transition made visible by T.

Since the simulation of the original process P is defined by the operational firing rules of CSP unrolling $T(P)$ while hiding the externalized hidden actions and the end events of visible actions, we do not go into details of unrolling the coordination process here. The important point is that the events are regarded as actions which are associated with UDFs and that these UDFs are executed between the atomic steps of the actions start and its end.

The following cases are to be considered when performing an action. The action either corresponds to a (a) synchronized event, (b) to a hidden event, (c) to a renamed event, or (d) to any other event.

The first case (a) is commonly considered to be a problem when matching actions with events. The argument against performing actions for synchronized events is the following question:

> "which of the synchronized components should perform the action?"

This question remained unanswered for some systems. For example the process analysis toolkit PAT [16] allows events – except synchronization events – to be associated with UDFs. Our answer to this question (and our argument in favor of performing actions for synchronized events) is that synchronization primarily affects the order of events and not their ownership. Thus, a single action is performed after the start event in an arbitrarily chosen context (either one of the threads performing the preceding actions, possibly another).

In the second case (b) the original event (being subject to hiding) defines the action to be performed. In the third case (c) the renamed event (not the original one) defines the action to be performed. Although this decision seems to be at odds with case (b), because both renaming and hiding can be regarded as a substitution of the original event's name (in a user-defined way or by τ respectively), the different treatments are necessary for uniquely identifying the UDFs to be executed. The reason is that hidden events cannot be synchronized on but renamed events are available for synchronization.

In the last case, the event directly identifies the UDF to be performed. It is noteworthy, however, that ✓ is not associated with a UDF. It solely models termination of a process, is commonly considered to be outside the alphabet of a process, and it is the only event not being split by T.

As an example, the single UDF to be performed by the following coordination process P (being algebraically equivalent to $SKIP$) is the one identified by event a.

$$P = ((b \rightarrow SKIP)[b \leftarrow a] \mid_{\{a\}} \mid a \rightarrow SKIP) \setminus \{a\}$$

This design decisions described above allow us to relinquish the idea that a sequential process models a single thread of control. A sequential process defines the order of its actions but these can be performed by different threads. This gives us great freedom in distributing the actions amongst processing units of the final system. Load balancing could, for example, be realized by statically creating a task queue and distributing the actions dynamically at runtime. Any of these informal descriptions above conforms to the formal definition of T.

4.2 Choosing Events

There must be some additional component added to the coordination environment dealing with the execution of hidden events. Visible events are available for external synchronization and chosen externally. But how to resolve conflicts of hidden actions?

Our solution is to use event listeners that reside above the component performing $T(P)$ and below the one realizing the hiding of externalized hidden actions and the end events of actions. This way, it can be ensured that the e and eh events are immediately performed so that $T(P)$ can proceed and offer any causally dependent s and sh events. The strategy how to choose amongst sh events, however, must be provided by the user of our coordination approach. Like the UDFs realizing the resolution of internal choices, listeners are to be injected into the coordination environment to deal with sh events introduced by hiding (those not being introduced by internal choice or timeout).

This leaves it open to the programmer to decide if hidden actions have priority over visible ones as, e. g., in the tau-priority model presented in [14], and how conflicts of hidden actions are resolved.

It is important to notice that this does not contradict non- invasiveness of our approach. By making hidden events on the outermost process available for internal event listeners, we provide a general way for resolving nondeterminism.

5 Detecting Data Races

CSP can be used as a coordination language independent of the target language. Thus, we do not consider a particular specification language for the UDFs here and present a purely mathematical proof obligation to ensure freedom of data races in a coordinated system. It is defined as a so-called *frame property*, based on our notion of possible concurrency (as presented in [8]), describing the modification behavior of a UDF with respect to another UDF. Refer to [11] for a more in-depth presentation of framing.

Let F be the type of all UDFs and Var the type of all modifiable entities (references to objects and primitive types). The mapping of events to UDFs is formally defined as a partial injective function such that its inverse is a total injection. It is

$$udf : \Sigma \rightarrowtail\!\!\!\!\!\!\rightarrow F\,, \qquad \text{and its inverse is} \qquad udf^{-1} : F \rightarrowtail \Sigma\,.$$

Hence, a UDF uniquely identifies an event but events do not necessarily identify a UDF. A *data race* is formally expressed using the following functions:

$$shared : F \times F \to \mathbb{P}\ Var$$

$$shared(f, g) \mathrel{\widehat{=}} (writes(f) \cap rw(g)) \cup (writes(g) \cap rw(f))$$

$$writes, reads, rw : F \to \mathbb{P}\ Var$$

$$rw(f) \mathrel{\widehat{=}} writes(f) \cup reads(f)$$

Two UDFs f and g suffer from a *data race* if their frames overlap and one modifies data also read or written by the other.

$$race(f, g) \mathrel{\widehat{=}} shared(f, g) \neq \emptyset$$

Let $conc : \Sigma \times \Sigma \to Bool$ be the predicate telling us whether or not two events are possibly concurrent in a given process P (i. e., $conc(x, y) \leftrightarrow (x, y) \in conc(P)$). This predicate is combined with udf^{-1} to the following predicate telling us whether or not two UDFs are possibly concurrent:

$$conc^F : F \times F \to Bool\,, \quad \text{where} \quad conc^F(f, g) \mathrel{\widehat{=}} conc(udf^{-1}(f), udf^{-1}(g))\,.$$

The proof obligation ensuring freedom of data races is

$$\forall f, g \in F : conc^F(f, g) \Rightarrow \neg\ race(f, g)\,.$$

A system violating this condition can be corrected by either adjusting the coordination process (by removing possible simultaneity) or modifying the UDFs (by making the frames distinct).

In general, determining the sets *reads* and *writes* of arbitrary UDFs is a hard problem (due to aliasing). Dealing with this issue is beyond the scope of the work presented here. However, it is noteworthy that specialized logics such as

separation logic [13] offer a prospective alternative for specifying (and verifying) properties such as the data independence of UDFs.

Besides this data related proof obligation, it must be proved that the UDFs are never called outside their preconditions and that they are guaranteed to terminate to obtain total correctness of a system. Furthermore, it must be proved that the UDFs resolving nondeterministic choices always return a valid process name.

As motivated in the Introduction, the sequential terminating UDFs implementing the sequential parts of the program are oblivious to concurrency modulo the proof obligation presented above. This means that our approach is noninvasive and allows us to turn a verified sequential system into a concurrent one by identifying parts of the program that make up the UDFs, adding a suitable CSP script and mapping events to UDFs. Provided that the additional proof-obligation can be discharged successfully, modular verification remains valid on that program. This implies that the sequential parts do not have to be modified at all. The coordination environment is the entry point to the final program and solely requires implementation of the mapping from events to UDFs.

6 Supported Processes

The approach presented here supports all finite alphabet CSP processes except the ill-formed recursive process $P = P$ (which is sometimes understood as *div*). This includes infinite state processes as well as diverging ones. The limitation to processes whose alphabet is finite matches the assumption that there are only finitely many UDFs assigned to the events of a process. Our approach to simulation is capable of dealing with any CSP process whose alphabet is finite. Consider the following divergent processes:

$$P = \mu\, P' \bullet (e \rightarrow P') \setminus \{e\} \qquad \text{and} \qquad Q = \mu\, Q' \bullet (e \rightarrow Q' \sqcap SKIP) \setminus \{e\}\,.$$

P models a process that runs forever without any interaction with its environment. This is not a problem in its own right, because, as shown in our case-study on modeling and implementing a workflow server in CSP [7], divergence may naturally arise when modeling server processes. Q may also diverge, but may as well eventually decide to terminate gracefully.

The reason for accepting such processes is that the internal actions resolving the choices can be used to implement local *fairness* conditions. The notion of fairness conditions refers to conditions that 'cure' processes that behave badly under the assumption that repeated (local) choices can always be resolved in the same way. The fact that this is unlikely to happen in reality is expressed by fairness. FDR does not support fairness but PAT explicitly deals with fairness, as described in [17]. Fairness can be specified on different levels of granularities (e. g., local or global) and can be regarded as an abstraction of probabilities.

Consequently, the process Q shown above is not necessarily diverging and might be guaranteed to terminate eventually under certain fairness assumptions. The same applies to other processes containing process control constructs such

as internal or external choice, timeout or interrupt allowing the process to eventually exit from cycles of hidden actions.

Both tools FDR and ProB can be used to verify that P unavoidably diverges while Q may eventually terminate. FDR proves this in the traces model because

$$\mathcal{T}[\![P]\!] = \{\langle\rangle\} \qquad \text{and} \qquad \mathcal{T}[\![Q]\!] = \{\langle\rangle, \langle\checkmark\rangle\}\,.$$

In the failures-divergences model $P = Q$ because both processes may diverge initially.

The same result can be obtained using the LTL model checking capabilities of ProB. In ProB's LTL syntax, the formula $\phi = F\ G\ [tau]$ states that a process unavoidably diverges (all of its executions eventually end up in an endless cycle of τ events). Now

$$P \models \phi \qquad \text{but} \qquad Q \not\models \phi\,.$$

The counterexample found by ProB expectedly shows that Q may eventually perform $\checkmark$. Another interesting example is the infinite state process

$$R = \mu\, R' \bullet (a \rightarrow R' \;|||\; b \rightarrow STOP)\,.$$

It can be simulated although it is obviously questionable if any reasonable program conforming to this design exists. Infinite state processes such as R are likely to eventually run out of memory, crashing the whole system.

Now, the same mechanism that can be used to implement fairness can be facilitated to turn theoretically infinite state processes into finite state processes. However, this requires knowledge of the structure of the processes when implementing the UDF resolving the internal choices. In the example of process R, the relevant UDF could count the a's and always choose the b if a certain bound is reached.

In this context it is important to observe that CSP_M scripts modeling infinite state systems cannot in general be checked by FDR because compilation of the script to the internal LTS will not terminate. The ProB LTL model checker, however, performs on-the-fly model checking and quite often succeeds in model checking infinite-state systems [12].

7 Implementation and Further Work

The coordination model presented here is given in a target language independent way. In this section, we present its implementation for the target language Java. It simulates a coordination process at runtime and executes user-defined Java code when performing an event. For now, it implements the operational semantics of the process operators *prefixing*, *sequential composition*, *internal choice*, *external choice*, *generalized parallel*, *timeout*, *interrupt*, *hiding*, and *renaming*.

The coordination environment provides the final class *CspEnvironment* encapsulating the coordination process. This class manages the events offered by the coordination process, listeners that deal with changes of offered events, and the mapping of events to UDFs. The mapping of events to UDFs is defined

by an implementation of the *CspEventExecutor* interface. The environment immutably references a single instance of a *CspProcessStore* holding the process configurations that describes the coordination process to be simulated. The environment instance is the only object that the implementation synchronizes on when performing the atomic start and end transitions modeling an event of the coordination process. The start transitions of visible events are available for external synchronization, while start transitions of hidden events are only accessible for event listeners. The end transitions are performed immediately after termination of the action's UDF.

The *CspEnvironment* provides a start method taking a process name that determines the coordination process instance. When started, the environment retrieves the coordination process from its store and adds the events offered by the process to its set of offered events. Then it informs the event listeners about that change. From that point on, interaction with the coordination environment is done by choosing events from the set of offered events and performing them. Subsequently the listeners are performed after every change of the set of events offered by the outermost process. This set changes only when an event is performed. If a client holds a reference to an event that is no longer available, it cannot be performed, of course (resulting in a no-op).

To create an executable system one has to instantiate a *CspEnvironment* with a *CspProcessStore* and a *CspEventExecutor*. One can then create a coordinated system as shown in Figure 1. A *CspSimulator* may be used to connect a *CspEnvironment* to the outside world. One useful example is the *SwingCspSimulator* that provides a simple Swing GUI to chose and perform events.

The example code shown in Figure 1 assumes the existence of suitable *CspEventExecutor* and *Filter* classes (filters are convenience objects helping event listeners to find the events that they act upon). The instances may be configured to fit the needs of the final system. Then a process store is created and must be filled with process configurations. A *CspEnvironment* is created using the *CspEventExecutor* and *CspProcessStore* instances. To deal with possibly hidden events, a *CspEventConsumer* takes the *Filter* instance as argument and is registered as a listener for event changes at the environment. Finally, the environment is run using a Swing GUI. Our implementation handles the nesting of process operators and the offered (initial) events of processes in an explicit way. This causes a considerable runtime overhead but allows us to resolve conflicts of actions when an action is started and to leave concurrent actions on offer while an action is performing.

The workflow server presented in [7] is built using this coordination environment. The workflow server accepts workflow definitions that are implemented in the same way and uses FDR to verify workflow definitions before activating them.

Extending the implementation towards the coordination of distributed systems is subject to future work. Explicit support for data storage and communication by the coordination environment is another concern that deserves further

```
CspEventExecutor cee = ... ;
Filter filter = ... ;
// setup cee and filter
CspProcessStore store = new CspProcessStore();
// register process configurations
CspEnvironment env = new CspEnvironment(store, cee);
env.registerListener(new CspEventConsumer(filter));
CspSimulator s = new SwingCspSimulator("my example", env);
s.run();
```

Fig. 1. Code stub of a coordinated Java program.

investigation. We also strive for a deeper integration with the functional verification of UDFs. Development of specialized provers for the conflict-freedom proof obligation is also subject to future work.

8 Related Work

Linda [2] is a parallel programming language that adds parallelism to sequential languages (e. g., C, Fortran, etc) and allows tasks to be distributed dynamically at runtime. Like Linda, our approach separates concurrency concerns from sequential ones in a machine- and language-independent way. In contrast to Linda, it builds on ordinary shared memory communication (as built into the underlying language) instead of a special memory model (the so-called *tuplespace*). Moreover, our approach is based on a formal method that is supported by a number of industrial-strength tools.

Reference Nets [9] provide an object-oriented High-Level Petri Net formalism where instances of Petri Nets carry references to other instances of Petri Nets instead of tokens. In that model, the firing of a transition moves a reference from one place to another. The concept of synchronous channels allows Petri Net instances to communicate when a transition fires. Reference Nets extend this to the execution of arbitrary (terminating) Java code. Thus, in that model, a Petri Net can be regarded as the coordination process of a system and the code attached to transitions relates to our understanding of actions. Unlike our approach, Reference Nets do not support verification of the coordination process and lack proof obligations relating the ordering of Petri Net transitions (as defined by their firing sequences) to the pre- and postconditions of their associated implementations, for example.

Another formal approach to the modeling and verification of distributed component based systems is described by Baier et al. [3]. Their approach is based on the model checker Vereofy which supports multiple input languages capable of modeling concurrent systems. However, their approach does not extend to implementing such systems but remains on the modeling level.

CSP++ [5] and JCSP [18] offer CSP-like channels as inter-thread communication facilities but realize only a limited set of CSP operators. The feature

that distinguishes our approach from CSP-based software libraries is that events are not interpreted as communications over a typed channel but as abstractions of arbitrary actions. It realizes the common understanding that an action can be modeled by a start and an end event (and the action executing between these two). Furthermore, the CSP model is simulated at runtime to drive the concurrent application.

Since sequential processes in CSP are viewed as executions of computational entities whose behavior can be observed in terms of events, it is natural to map a single sequential process to its own thread. This is the general idea underlying CSP++ and JCSP. Our approach does not necessarily map two causally dependent threads to the same thread. It merely maintains the order of UDFs as defined by the events of the CSP script coordinating the system. This is comparable to Linda's feature of distributing tasks at runtime.

9 Conclusions

In this paper, our approach to using CSP as a coordination language is presented. It unravels the various abstractions that are carefully built into CSP for the purpose of coordinating concurrent systems in a noninvasive way. The approach taken here is quite different from other approaches to implementing CSP because it does not built on the channel concept which is a common approach realizing the hand-shaken communication. Instead, our approach is based on atomic steps denoting start- and endpoints of a system's actions. It also deals with internal (hidden) actions. Associating the resolution of internal choices with UDFs undoes nondeterminism inherent to the CSP model coordinating the program. This allows us, for example, to turn systems that are not fair in the standard CSP semantics into fair ones.

The semantic foundation of the coordination runtime environment is given by a syntactic process transformation turning a standard interleaving CSP process into a truly concurrent one. The transformation also allows us to predict which functions are possibly executed in parallel. Dependent on possibly concurrent events we stated a proof obligation for ensuring absence of data races.

Our approach allows us to separate concurrency issues from sequential ones, and to reuse the concurrent design of the system at runtime. It also enables us to use state-of-the-art CSP tools such as FDR [14] and ProB [10] for the automated verification of concurrency aspects of the software, and integrates with modular verification of UDFs. The use of CSP makes it especially well suited for systems with highly communicative concurrent components.

Compared to other CSP implementations, the approach presented here supports a richer CSP in terms of supported process operators. Unlike other CSP library-level implementations, our approach is noninvasive. It allows the system's basic actions to be coordinated very directly. Our approach is capable of handling infinite state and divergent processes. Compared to other coordination languages, ours has the advantage of its strong integration with formal methods.

However, this comes at the cost of the runtime overhead of explicitly managing processes and events caused by simulation of CSP.

References

1. J. Abrial. *The B Book - Assigning Programs to Meanings.* Cambridge University Press, August 1996.
2. S. Ahuja, N. Carriero, and D. Gelernter. Linda and Friends. *Computer*, 19(8):26 –34, August 1986.
3. C. Baier, T. Blechmann, J. Klein, and S. Klüppelholz. A Uniform Framework for Modeling and Verifying Components and Connectors. In J. Field and V. Vasconcelos, editors, *Coordination Models and Languages*, volume 5521 of *Lecture Notes in Computer Science*, pages 247–267. Springer, 2009.
4. M. Barnett, B. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *FMCO*, pages 364–387, 2005.
5. W. B. Gardner. Converging CSP specifications and C++ programming via selective formalism. *ACM Trans. Embed. Comput. Syst.*, 4(2):302–330, 2005.
6. C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall International, 1985.
7. M. Kleine and T. Göthel. Specification, Verification and Implementation of Business Processes using CSP. In *4th IEEE International Symposium on Theoretical Aspects of Software Engineering*, pages 145–154. IEEE Computer Society, 2010.
8. M. Kleine and J. W. Sanders. Simulating truly concurrent CSP. In *Brazilian Symposium on Formal Methods (SBMF 2010)*. Springer, 2010.
9. O. Kummer. *Referenznetze.* Logos Verlag, Berlin, 2002.
10. M. Leuschel and M. Fontaine. Probing the Depths of CSP-M: A new FDR-compliant Validation Tool. In *International Conference on Formal Engineering Methods*, pages 278–297. Springer, 2008.
11. P. Müller. *Modular specification and verification of object-oriented programs.* Springer-Verlag, Berlin, Heidelberg, 2002.
12. D. Plagge and M. Leuschel. Seven at one stroke: LTL model checking for High-level Specifications in B, Z, CSP, and more. *STTT*, 2008.
13. J. Reynolds. Separation logic: a logic for shared mutable data structures, 2002.
14. A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, 2005.
15. S. Schneider and H. Treharne. Verifying Controlled Components. In *IFM*, pages 87–107, 2004.
16. J. Sun, Y. Liu, and J. S. Dong. Model Checking CSP Revisited: Introducing a Process Analysis Toolkit. In *International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, pages 307–322. Springer, 2008.
17. J. Sun, Y. Liu, J. S. Dong, and J. Pang. PAT: Towards Flexible Verification under Fairness. *Proceedings of the 21th International Conference on Computer Aided Verification (CAV'09)*, 5643:709–714, 2009.
18. P. H. Welch. Process Oriented Design for Java: Concurrency for All. In H. R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000)*, volume 1, pages 51–57. CSREA, CSREA Press, June 2000.