Synthesizing Concurrency Control Components from Process Algebraic Specifications

Edoardo Bontà[1], Marco Bernardo[1], Jeff Magee[2], and Jeff Kramer[2]

[1] Istituto di Scienze e Tecnologie dell'Informazione, Università di Urbino
[2] Department of Computing, Imperial College London

Abstract. Process algebraic specifications can provide useful support for the architectural design of software systems due to the possibility of analyzing their properties. In addition to that, such specifications can be exploited to guide the generation of code. What is needed at this level is a general methodology that accompanies the translation process, which in particular should help understanding whether and when it is more appropriate to implement a software component as a thread or as a monitor. The objective of this paper is to develop a systematic approach to the synthesis of correctly coordinating monitors from arbitrary process algebraic specifications that satisfy some suitable constraints. The whole approach will be illustrated by means of the process algebraic specification of a cruise control system.

1 Introduction

Although process algebras were originally conceived as a means for producing abstract views of concurrent programs and reasoning about their properties [13, 9, 3], due to their compositional nature it was soon realized their adequacy for modeling complex systems [6]. More recently process algebras have been integrated within the software architecture design level [14, 15], because they provide support for the early assessment of the gross system properties. This has resulted in a family of process algebraic ADLs, for which several techniques based on equivalence checking have been developed for the component-oriented verification and diagnosis of architectural mismatch freedom [2, 12, 11, 10, 7, 1].

At the software architecture design level, process algebras have turned out to be useful also for code generation purposes. In [12] it is shown how a disciplined process algebraic modeling is beneficial at subsequent stages for guiding the implementation of Java software. In [4, 5] an automatic code generator is presented, which translates process algebraic architectural descriptions into multithreaded Java programs on the basis of a transparent Java package called Sync that ensures the correct thread synchronization.

In a process algebraic description, the behavior of a software component is specified through a sequence of action-based equations, which define possibly alternative execution traces composed of local actions and actions interacting with other components. In this framework two natural candidates for the target

of the translation of the process algebraic description of a component are a thread and a monitor.

What is needed at this level is a general methodology that accompanies the translation process, which in particular should help understanding whether and when it is more appropriate to implement a software component as a thread or as a monitor. This would overcome some limitations that are present in the current process algebraic approaches. With respect to [12] generality would be gained, as it would become possible to undertake the translation of arbitrary process algebraic descriptions. With respect to [4, 5], where only threads are taken into account, the performance of the generated code may be improved thanks to the synthesis of monitors as they would reduce the thread context switch frequency. Moreover the presence of monitors would result in a lightweight concurrency control management with respect to package Sync, with the monitors themselves constituting explicit coordination areas that were not available before to the developer adopting the approach of [4, 5].

The objective of this paper is to develop a systematic approach to the synthesis of correctly coordinating Java monitors from arbitrary process algebraic component descriptions that satisfy some suitable constraints. The constraints are related to the fact that a monitor is a passive entity, which typically encapsulates data in a way that guarantees a mutually exclusive access. In other words, a monitor coordinates the access of the threads to its methods, but its statements are executed by the entering threads. As we shall see, in order to enforce a correct concurrency control when using a monitor, it is sufficient that a thread taking the control of the monitor can perform neither interacting actions nor infinitely many local actions while inside the monitor.

Once these constraints are satisfied, the process algebraic description of a component can systematically be transformed into a canonical form that we call monitor normal form, from which it is easy to synthesize a Java monitor. The constraints and the approach will be illustrated by means of the process algebraic specification of a cruise control system taken from [12], which will be used as a running example throughout the paper.

This paper is organized as follows. In Sect. 2 we present the constraints that guarantee the derivability of a monitor from a process algebraic component description. In Sect. 3 we show how to transform into monitor normal form a process algebraic component description that satisfies all the constraints. In Sect. 4 we describe how to synthesize a Java monitor from a monitor normal form. Finally, in Sect. 5 we provide some remarks on related and future work.

2 Monitor Constraints

In this section we present a set of constraints under which it is possible to synthesize a correctly coordinating monitor from the process algebraic description of a software component. Before doing so, we introduce some terminology and we recall the way in which threads and monitors interact with each other in an object-oriented language like Java.

2.1 Terminology

In our process algebraic view, both thread and monitor classes should be modeled as architectural element types [1]. An architectural element type representing a Java class that extends or implements a thread base class will be called *native-thread type* and will be translated into a *native-thread component*. An architectural element type representing a Java monitor class will instead be called *monitor type* and will be translated into a *monitor component*.

Furthermore, we distinguish between two kinds of interacting actions, which we simply call interactions from now on. An *active-control interaction* is performed by an architectural element whenever it starts communicating with another architectural element. A *passive-control interaction* is executed by an architectural element whenever it is waiting for another architectural element to start communicating with it. In particular the entry points (and hence implicitly the exit points) of the monitor types will be described through passive-control interactions.

2.2 Thread-Monitor Interaction

Given a native-thread component T and a monitor component M, the interaction between them takes place by means of the component control switch depicted in the sequence diagram of Fig. 1. When T intends to interact with M, T invokes a synchronized method of M – which corresponds to performing an active-control interaction – so that the thread t leaves T and waits until M is ready to interact.

More precisely, in a synchronous model t waits outside M if another thread is currently running inside M, otherwise it immediately enters M and possibly blocks, which happens when t has to wait for a notification related to a condition synchronization of M that does not hold upon entering M. In an asynchronous model, instead, an exception is raised if a condition synchronization for t does not hold and either there is no thread running inside M or the thread currently running inside M leaves it without notifying such a condition synchronization. We recall from [12] that a condition synchronization permits a monitor to block threads until a particular condition holds, such as e.g. a count becoming non-zero, a buffer becoming empty, or new input becoming available.

When M is ready, t takes its control and executes a sequence of statements of M corresponding to local actions, at the end of which t possibly notifies one of the threads blocked inside M about the validity of a condition synchronization and leaves the monitor. The end of such a statement sequence coincides either with the monitor termination or with the execution of the last local action before a passive-control interaction.

In order to achieve a correct concurrency control, it suffices that the thread taking the control of the monitor executes finitely many statements without moving to another monitor or invoking a method of another thread before leaving the monitor in which it is running [3]. In this way a thread will stay within the

[3] Note that this does not prevent the monitor from invoking methods of the Java library and creating new non-thread objects.

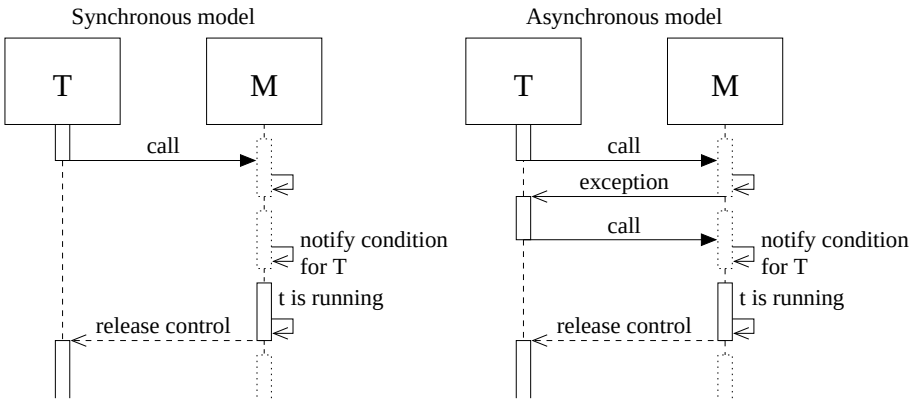

Fig. 1. Component control switch from native-thread T to monitor M

monitor for a finite amount of time (up to possible condition synchronizations that will never hold), and will not cause any interference between the monitor and other monitors.

From the considerations above, we derive that the avoidance of (i) endless executions of local actions and (ii) active-control interactions guarantees that a correctly coordinating monitor can be obtained from the process algebraic specification of a component. For the sake of completeness, a third technical constraint, related to the avoidance of (iii) non-disjoint hybrid choices between sets of local actions and sets of interactions, must be satisfied.

2.3 Constraint 1: No Endless Executions of Local Actions

Since a monitor is a passive entity that coordinates other components, it is desirable that a thread taking the control of the monitor runs inside the monitor only for a finite amount of time. In the worst case, it may happen that the thread blocks forever inside the monitor because of a condition synchronization that will never hold. However, this does not prevent other threads from entering the monitor and running.

In order to achieve finiteness, we need to enforce that the maximum number of consecutive local actions that can be performed inside a candidate monitor type is finite. This can easily be checked on the process algebraic description of a candidate monitor type by verifying the absence of cycles of local actions.

As we shall see in Sect. 4, each local action will be translated into a method to be manually filled in later on. If we adhere to the guidelines of [5], according to which non-terminating statements should be avoided within these methods, a finite sequence of local actions will be executed in a finite amount of time. In this way the absence of cycles of local actions proved at the process algebraic level is guaranteed to be preserved at the Java code level.

2.4 Constraint 2: No Active-Control Interactions

A monitor coordinates other components but should not invoke methods of other monitors or threads. Therefore, a candidate monitor type should not possess any active-control interaction. This can trivially be verified at the level of the process algebraic description if this is suitably annotated with information about the control flow direction (like e.g. in PADL [1]).

The reason for this constraint is to prevent a thread running inside a monitor from moving to another monitor or invoking a method of another thread before finishing its execution within the monitor in which it is running. This constraint thus implies that interferences among monitors are avoided and that any monitor component can passively interact only with thread components. In particular, deadlock cannot occur because of a possible invocation of methods belonging to the same thread that is currently running inside the monitor. Moreover, this constraint ensures, together with the previous one, that a thread runs inside a monitor only for a finite amount of time.

Note that this constraint does not prevent a monitor component from invoking methods of the Java library and creating new non-thread objects. In fact, the methods translating the local actions of the monitor component are free to create local objects and to interact with them. However, this should not alter the topology prescribed by the process algebraic architectural description.

2.5 Constraint 3: No Non-Disjoint Hybrid Choices

A *hybrid choice* in the process algebraic description of a component is a choice between a non-empty set of interactions and a non-empty set of local actions. The problem with hybrid choices is that they may hamper the detection of the action sequence corresponding to the statement sequence that should be executed by a thread running inside a monitor.

In fact, recalled that the monitor entry and exit points are described through passive control interactions, to automatically detect the beginning and the end of the action sequence in a candidate monitor type we need that the sequence is comprised between two passive-control interactions. A choice between a passive-control interaction and a local action would make it impossible to decide whether the currently running thread has completed its task or not, unless the two actions are preceded by two disjoint conditions.

As a consequence of this constraint, a candidate monitor type can contain only choices among all interactions or all local actions. This can easily be checked at the process algebraic description level. In addition, hybrid choices are admitted provided that the involved actions are preceded by disjoint boolean conditions, i.e. the logical conjunction of the condition of any involved interaction and the condition of any involved local action must be false.

3 Syntactic Transformation into Monitor Normal Form

Once the three constraints defined in the previous section are satisfied by the process algebraic description of a candidate monitor type, it is possible to proceed

to the transformation of the description itself into monitor normal form. Starting from this canonical form, it will be possible to straightforwardly synthesize the Java implementation of the monitor type.

In order to facilitate the derivation of each method of the targeted Java monitor class, a good idea may be to rewrite the process algebraic specification of the monitor type in such a way that all the interacting actions are collected into a single equation. Due to constraint 2, each such interaction is a passive-control one, so if we place all of them at the beginning of a different branch of a choice, we exactly characterize the point at which the monitor is waiting for a thread to take its control.

The process algebraic specification in monitor normal form obtained at the end of the rewriting process will be formed by:

- An *interacting choice equation*, which is composed of a choice in which every branch starts with an interaction possibly followed by local actions only.
- A group of *local equations*, which are original equations of the monitor type that include only local actions.
- A group of *setting equations*, which are the original non-local equations whose branches that have been moved to the interacting choice equation are replaced by an invocation of the latter equation with suitably set parameters.

This monitor normal form can be achieved through a sequence of five steps, which will be exemplified on the process algebraic description of a cruise control system taken from [12].

3.1 Example: A Cruise Control System

An automobile cruise control system is governed by means of three buttons – `on`, `off`, and `resume` – and takes into account two pedals – `accelerator` and `brake`. When the engine is running and `on` is pressed, the cruise control system records the current speed and maintains the car at this speed. When `accelerator`, `brake` or `off` is pressed, the cruise control system disengages but retains the speed setting. If `resume` is pressed, the system accelerates or de-accelerates the car back to the previously-recorded speed.

The kernel of the cruise control system is provided by a cruise controller, which includes a speed control that is initially disabled. While the latter clears and records the speed setting and, when enabled, sets the throttle according to the current speed and the recorded speed, the behavior of the former is more complex. When the engine is switched on (`engineOn`), speed clearing is triggered (`clearSpeed`) and the cruise controller becomes active. When active, pressing `on` triggers the recording of the current speed (`recordSpeed`) and enables the speed control (`enableControl`). The system is then cruising. Pressing `on` again triggers the recording of the new current speed and the system remains cruising. Pressing `off`, `brake` or `accelerator` disables the speed control (`disableControl`) and sets the system to standby, from which the system can return to the cruising state whenever `resume` or `on` is pressed. Switching the engine off (`engineOff`) at any time makes the cruise controller inactive and the speed control disabled.

We now provide the FSP specification [12] of the cruise controller:

```
INACTIVE = (engineOn->clearSpeed->
               (engineOff->INACTIVE
               |on->recordSpeed->enableControl->CRUISING
               )
           ),
CRUISING = (engineOff->disableControl->INACTIVE
           |{off,brake,accelerator}->disableControl->STANDBY
           |on->recordSpeed->enableControl->CRUISING
           ),
STANDBY  = (engineOff->INACTIVE
           |resume->enableControl->CRUISING
           |on->recordSpeed->enableControl->CRUISING
           ).
```

where:

- `INACTIVE`, `CRUISING`, and `STANDBY` are the names of the three process algebraic equations that formalize the behavior of the cruise controller.
- `engineOn`, `engineOff`, `on`, `off`, `brake`, `accelerator`, and `resume` are the interactions.
- `clearSpeed`, `recordSpeed`, `enableControl`, and `disableControl` are the local actions.
- The symbol "`->`" is the action prefix operator: $\{a_1, \ldots, a_n\}$ `->` P executes an action from the set and then behaves as P.
- The symbol "`|`" is the choice operator: P_1 `|` P_2 behaves as either P_1 or P_2.

If `engineOn`, `engineOff`, `on`, `off`, `brake`, `accelerator`, and `resume` are considered as passive-control interactions, it is not difficult to observe that all the three monitor constraints defined in Sect. 2 are satisfied by the FSP description of the cruise controller.

3.2 Step 1: Rewriting Complex Choices

If the process algebraic specification of a monitor type contains some choices among interactions that are written in an abbreviated notation, such choices must be expanded. Likewise, if the specification contains some nested choices among interactions, such choices must be flattened. By doing so, it will be easier to handle the branches of the choices among interactions as we shall see in the subsequent steps.

In the FSP specification of the cruise controller, the equations `INACTIVE` and `STANDBY` do not contain complex choices, while the equation `CRUISING` contains the abbreviated FSP notation $\{\triangleleft$*action list*$\triangleright\}$, hence it is expanded into:

```
CRUISING = (engineOff->disableControl->INACTIVE
           |off->disableControl->STANDBY
           |brake->disableControl->STANDBY
           |accelerator->disableControl->STANDBY
           |on->recordSpeed->enableControl->CRUISING
           )
```

3.3 Step 2: Splitting the Equations

Since in the interacting choice equation of the monitor normal form any branch must start with an interaction, every interaction or choice among interactions that does not occur at the beginning of the body of an equation must be moved together with what follows it into a new equation. The moved block is replaced in the original equation by an invocation of the new equation. At the end of this splitting process, any interaction will be at the beginning of some equation.

In the FSP specification of the cruise controller, only the first equation needs to be transformed, because in the equations `CRUISING` and `STANDBY` all the occurrences of interactions are already at the beginning of some branch. The equation `INACTIVE` thus becomes:

```
INACTIVE         = (engineOn->clearSpeed->SPLIT_1_INACTIVE),
SPLIT_1_INACTIVE = (engineOff->INACTIVE
                   |on->recordSpeed->enableControl->CRUISING
                   )
```

3.4 Step 3: Building the Interacting Choice Equation

The interacting choice equation can now be built by suitably merging into a single equation the equation body branches that start with an interaction.

In order to preserve the semantics of the original equations of the monitor type, the resulting interacting choice equation needs several parameters representing the current interacting state of the monitor. Such a state can be encoded through the non-local equation (among the ones present at the end of step 2) describing the current behavior – bounded integer parameter `eq` – and the set of interactions that are currently enabled – boolean parameters `g_` representing guards associated with the enabledness of the interactions. Note that parameter `eq` is strictly necessary because the same set of interactions may be enabled in several different non-local equations. On the other hand, the guards `g_` are necessary to decide the branch to be undertaken in the current interacting state and, as we shall see, useful to implement the condition synchronizations.

The body of the interacting choice equation is thus a guarded choice among all the merged equation body branches. In particular, if one of the involved bodies started with a single interaction, the whole body becomes a branch of the interacting choice equation. Instead, if it started with a choice among all interactions, each branch of such a choice becomes a branch of the interacting choice equation. Finally, if it started with a disjoint hybrid choice, only the branches starting with an interaction move to the interacting choice equation [4].

Each branch of the interacting choice equation is preceded by a boolean expression composed of the logical conjunction of: the control that the value of `eq` corresponds to the value associated with the non-local equation body that contained the considered branch, the guard `g_` associated with the first

[4] We shall see later on that this does not disrupt the semantics of the disjoint hybrid choice, hence of the original process algebraic specification.

interaction of the branch itself, and other possible conditions inherited from the original branch.

In the FSP specification of the cruise controller, the bodies of the equations INACTIVE, SPLIT_1_INACTIVE, CRUISING, and STANDBY are represented by the values 0, 1, 2, and 3 of parameter eq, respectively, and their branches are merged into the following interacting choice equation:

```
INTER_CH_EQ[eq:0..3]        [g_engineOn:Boolean]   [g_engineOff:Boolean]
           [g_on:Boolean]   [g_off:Boolean]        [g_resume:Boolean]
           [g_brake:Boolean][g_accelerator:Boolean]  =
  (when((eq==0) && g_engineOn)  engineOn->clearSpeed->SPLIT_1_INACTIVE
  |when((eq==1) && g_engineOff)                    engineOff->INACTIVE
  |when((eq==1) && g_on)      on->recordSpeed->enableControl->CRUISING
  |when((eq==2) && g_engineOff)    engineOff->disableControl->INACTIVE
  |when((eq==2) && g_off)                 off->disableControl->STANDBY
  |when((eq==2) && g_brake)             brake->disableControl->STANDBY
  |when((eq==2) && g_accelerator) accelerator->disableControl->STANDBY
  |when((eq==2) && g_on)      on->recordSpeed->enableControl->CRUISING
  |when((eq==3) && g_engineOff)                    engineOff->INACTIVE
  |when((eq==3) && g_resume)           resume->enableControl->CRUISING
  |when((eq==3) && g_on)      on->recordSpeed->enableControl->CRUISING
  )
```

3.5 Step 4: Rewriting Non-Local Equations into Setting Equations

The interacting choice equation built in step 3 does not replace the original equations. This refers not only to local equations and equations with disjoint hybrid choices – which are not completely involved in the construction of the interacting choice equation – but also to the other equations, as invocations to them are still around.

The body of each non-local equation is thus rewritten in such a way that its possible local branches are preserved. By contrast, its branches that have been moved to the interacting choice equation are replaced by a single invocation of the latter equation with suitably set actual values for parameters eq and g_. So, we refer to such a rewritten equation as a setting equation.

The actual value of eq passed to the interacting choice equation has to be the value associated with the setting equation body. The actual values of the boolean guards are set as follows. If an interaction does not occur at the beginning of any moved branch of the original non-local equation, then the corresponding guard g_ is set to false. If it occurs instead and at least one of its occurrences was not guarded by any condition in the original branch that contained it, the corresponding guard g_ is set to true. Finally, if it occurs and all of its occurrences were guarded by some condition in the original branches that contained them, the corresponding guard g_ is set to the logical disjunction of these conditions (if at least one of them holds true, then the interaction is enabled).

If an original non-local equation started with a single interaction or a choice among interactions only, its body is entirely replaced by an invocation of the interacting choice equation having the above-mentioned actual values for eq

and g_. In the case in which the original equation contained a disjoint hybrid choice, instead, its body preserves all the local branches. The other (interacting) branches, moved together with their conditions to the interacting choice equation, are replaced by a single branch. This branch contains the invocation of the interacting choice equation preceded by the logical disjunction of all the conditions associated with the interacting branches. In this way the semantics of the selection between the group of local branches and the group of interacting branches is preserved, with the selection within the latter group being deferred to the interacting choice equation.

In the FSP specification of the cruise controller, the non-local equations INACTIVE, SPLIT_1_INACTIVE, CRUISING, and STANDBY are rewritten into the following setting equations:

```
INACTIVE =
  INTER_CH_EQ[0] [True] [False][False][False][False][False][False],
SPLIT_1_INACTIVE =
  INTER_CH_EQ[1] [False][True] [True] [False][False][False][False],
CRUISING =
  INTER_CH_EQ[2] [False][True] [True] [True] [False][True] [True],
STANDBY =
  INTER_CH_EQ[3] [False][True] [True] [False][True] [False][False]
```

3.6 Step 5: Rearranging the Interacting Choice Equation

As a final step, the interacting choice equation undergoes to a sorting of its branches as well as to a number of optimizations. On the one hand, the branches are lexicographically sorted on the basis of their guards g_ associated with their starting interactions. The reason is that all the branches starting with the same interaction will be translated into a single synchronized method of a Java monitor class, hence this sorting should facilitate the code generation.

On the other hand, some optimizations are useful to simplify the structure of the interacting choice equation and thus of the monitor to be synthesized. First, if an interaction occurs at the beginning of only one of the branches associated with a same value of eq, in that branch the possible condition inherited from the original branch can be removed. In fact, the same condition is already contained in the guard g_ associated with the considered branch.

Second, if all the branches with the same initial interaction are associated with a single value of eq, the check on eq can be removed from these branches. In fact, this means that the initial interaction was present only in a single non-local equation body of the original specification, and the guard g_ associated with the action can be true only when eq has that value.

Third, if several branches are identical up to their boolean expressions – i.e. checks on different values of eq and possibly other different conditions inherited from the original specification – these branches can be collapsed into a single one. This new branch is preceded by an expression which includes, besides the checks on g_ and the different values that eq can take on, the logical disjunction of the conditions of the collapsed branches. If the interaction occurs only at the

beginning of such a new branch, by virtue of the first two optimizations the disjunction of the inherited conditions and the check on `eq` can be removed.

In the FSP specification of the cruise controller, the second optimization can be applied to the branches starting with `engineOn`, `off`, `brake`, `accelerator`, and `resume`. The third optimization can be applied to the branches beginning with `engineOff` and corresponding to the values 1 and 3 of `eq`, and to the branches beginning with `on` and corresponding to the values 1, 2, and 3 of `eq`. In the latter case the check on the different values of `eq` can be removed. After applying such optimizations, the interacting choice equation becomes:

```
INTER_CH_EQ[eq:0..3]        [g_engineOn:Boolean]    [g_engineOff:Boolean]
           [g_on:Boolean]   [g_off:Boolean]         [g_resume:Boolean]
           [g_brake:Boolean][g_accelerator:Boolean]  =
  (when(g_engineOn)             engineOn->clearSpeed->SPLIT_1_INACTIVE
  |when(g_engineOff && ((eq==1) || (eq==3)))       engineOff->INACTIVE
  |when(g_engineOff && (eq==2))    engineOff->disableControl->INACTIVE
  |when(g_on)                 on->recordSpeed->enableControl->CRUISING
  |when(g_off)                           off->disableControl->STANDBY
  |when(g_brake)                       brake->disableControl->STANDBY
  |when(g_accelerator)           accelerator->disableControl->STANDBY
  |when(g_resume)                     resume->enableControl->CRUISING
  )
```

3.7 Correctness of the Transformation

The syntactic transformation of the process algebraic description of a monitor type into monitor normal form is correct in the following sense.

Theorem 1. *Let M be the process algebraic description of a monitor type and let M' be the process algebraic description of the monitor normal form obtained by applying to M the syntactic transformation. Then the LTS underlying M' is isomorphic to the LTS underlying M.* ■

4 Monitor Implementation

The application of the steps illustrated in the previous section allows an arbitrary process algebraic description of a monitor type to be rewritten into its semantically equivalent monitor normal form. In this section we show how to synthesize a monitor component as a Java class from a monitor normal form.

In the Java monitor class, the interacting choice equation will be translated into a set of public synchronized methods each corresponding to a different interaction. Instead, the setting and local equations will be translated into non-public methods of the monitor. Finally, the constructor of the Java monitor class will invoke the method corresponding to the first equation of the process algebraic description.

The synthesis of the monitor will be exemplified below by translating into Java code the monitor normal form of the process algebraic description of the cruise controller. This is accomplished through a sequence of four steps, which guide the automated generation of the Java code.

4.1 Translating Local Actions into Stub Class Methods

On the basis of the approach proposed in [5], each local action of the process algebraic description of a monitor type will be synthesized in the Java monitor class as an invocation of a non-completely specified public method of an auxiliary class, which we call *stub class*. In this way the software developer can subsequently fill in the methods associated with the local actions, without any intervention on the main monitor class. The stub class will be instantiated by the constructor of the Java monitor class.

Recalled that the FSP specification of the cruise controller contains the local actions `clearSpeed`, `recordSpeed`, `enableControl`, and `disableControl`, the related Java stub class `LocalActionsController` is synthesized as follows:

```
class LocalActionsController {
  public LocalActionsController() {/* FILL IN THE CONSTRUCTOR BODY */}
  public void clearSpeed()        {/* FILL IN THE METHOD BODY */}
  ...
  public void disableControl()    {/* FILL IN THE METHOD BODY */}
}
```

4.2 Synthesizing the Monitor Class Constructor

The first-executed method of the Java monitor class, i.e. the constructor, is in charge of the instantiation of the stub class for the local actions and of the invocation of the method corresponding to the (local or setting) equation of the monitor normal form associated with the first equation of the original process algebraic description.

Besides the definition of the constructor, at the beginning of the monitor class there is the declaration/definition of some private members. The first private member is an object of the stub class for the local actions that will be instantiated by the constructor. Then, an integer variable `eq` and a boolean array `guard[]` are declared, which translate the parameters of the interacting choice equation, together with the definition of some integer constants associated with the setting equations, which represent the values that `eq` can take on.

Referring to the monitor normal form of the cruise controller, the Java monitor class starts as follows:

```
private LocalActionsController laController;
private int eq;
private boolean guard[];
private final static int INACTIVE         = 0,
                         SPLIT_1_INACTIVE = 1,
                         CRUISING         = 2,
                         STANDBY          = 3;
public Controller() {
  laController = new LocalActionsController();
  inactive();
}
```

4.3 Translating Setting and Local Equations

The setting and local equations of the monitor normal form are translated into non-public methods of the Java monitor class. Since these equations do not contain interactions, only sequences of/choices among local actions have to be considered during their translation.

While a sequence of local actions can easily be synthesized as a sequence of invocations of the associated stub methods, a choice among local actions has to be treated carefully. In fact, even if the branches of the choice can be guarded by some conditions, it is not necessarily the case that such conditions are disjoint. One possibility is to translate such nondeterministic choices by means of the selection statements provided by Java, with the software developer subsequently removing nondeterminism at the code level. Another possibility is to synthesize a probabilistic mechanism to randomly select a branch whose associated condition holds true. This solution may be appropriate for the implementation of simulation software and randomized concurrent algorithms.

An invocation of a setting or local equation is turned into an invocation of the monitor class method translating the equation itself. Each setting equation contains in turn an invocation of the interacting choice equation, which corresponds to the fact that the thread currently running inside the monitor is on the verge of leaving it. This invocation is translated into a sequence of assignment statements in which `eq` and `guard[]` are set to the corresponding actual parameters specified in the invocation. Since before leaving the monitor the thread has to notify the other threads possibly blocked inside the monitor, the assignment statement sequence is followed by an invocation of the Java method `notifyAll()` to wake up all the threads waiting inside the monitor. The unblocking conditions, which have just been updated by setting `guard[]`, will be handled by the synchronized methods translating the interacting choice equation.

Referring to the monitor normal form of the cruise controller, the translation of the setting equation `INACTIVE` is implemented as follows:

```
protected synchronized void inactive() {
  eq = INACTIVE;
  guard = new boolean[] {true, false, false, false, false, false, false};
  notifyAll();
}
```

4.4 Translating the Interacting Choice Equation

Any group of branches of the interacting choice equation that start with the same interaction is translated into a public synchronized method of the monitor class. The resulting methods basically translate the communication of the passive-control interactions of the monitor type with the active-control interactions of native-thread types to which the passive-control ones are attached.

At the beginning of each such method, the boolean guard associated with the related interaction is translated into a condition synchronization statement:

```
while (!<guard>)
  wait();
```

If the boolean guard is true, a thread can enter the monitor without blocking. Otherwise it blocks on the Java method `wait()` until another thread leaves the monitor by setting the guard to true and notifying about this event.

The condition synchronization is implemented in a different way whenever the related interaction is asynchronous. The reason is that in this case, if the condition synchronization is false, an entering thread has to exit the monitor without blocking. This is implemented as follows:

```
if (!<guard>)
  throw new AsyncInteractionNotReadyException();
```

After the condition synchronization, within the method associated with an interaction we have an `if-else` statement, which handles the selection among the branches (starting with the considered interaction) based on the value of `eq`. For those branches sharing the same value of `eq` a nested selection statement is necessary, which is based on inherited conditions.

Referring to the monitor normal form of the cruise controller, the branches of the interacting choice equations starting with `engineOff` are translated into the following method (index 1 of `guard[]` is associated with `engineOff`):

```
public synchronized void engineOff()
    throws AsyncInteractionNotReadyException {
  if (!guard[1])
    throw new AsyncInteractionNotReadyException();
  if ((eq == SPLIT_1_INACTIVE) || (eq == STANDBY))
    inactive();
  else /* if (eq == CRUISING) */ {
    laController.disableControl();
    inactive();
  }
}
```

5 Conclusion

In this paper we have addressed the problem of synthesizing concurrency control components, in the form of Java monitor classes, from arbitrary process algebraic specifications. The problem of synthesizing Java monitors has been previously addressed in [16, 8] outside the process algebra field.

In [16] a tool equipped with a model checker automatically generates Java monitor classes from monitor descriptions written in Action Language. The correctness of the synchronization and of the behavior of the generated Java monitor is guaranteed by construction, independently of the context of the monitor description. Unlike our approach, this approach requires that the monitor description conforms a priori to a specific monitor template.

In [8] implementations of synchronization policies are generated in Java through synchronized methods and lock objects. While in the previously described approaches the generated Java monitors are obtained from formal specifications and are correct by construction, in this approach the code is generated from critical regions delimited by the developer with high-level synchronization

directives and the correctness of the implemented synchronization policies is verified at the code level via model checking.

For the future we plan to conduct further investigations on the monitor constraints, in particular with respect to specific contexts, and to develop a tool – to be hopefully integrated inside the automatic code generator PADL2Java [4, 5] – that synthesizes a Java monitor class from any process algebraic specification that satisfies the three constraints.

References

1. A. Aldini and M. Bernardo, *"On the Usability of Process Algebra: An Architectural View"*, in Theoretical Computer Science 335:281-329, 2005.
2. R. Allen and D. Garlan, *"A Formal Basis for Architectural Connection"*, in ACM Trans. on Software Engineering and Methodology 6:213-249, 1997.
3. J.A. Bergstra, A. Ponse, and S.A. Smolka (eds.), *"Handbook of Process Algebra"*, Elsevier, 2001.
4. M. Bernardo and E. Bontà, *"Generating Well-Synchronized Multithreaded Programs from Software Architecture Descriptions"*, in Proc. of the *4th Working IEEE/IFIP Conf. on Software Architecture (WICSA 2004)*, IEEE-CS Press, pp. 167-176, Oslo (Norway), 2004.
5. M. Bernardo and E. Bontà, *"Preserving Architectural Properties in Multithreaded Code Generation"*, in Proc. of the *7th Int. Conf. on Coordination Models and Languages (COORDINATION 2005)*, LNCS 3454:188-203, Namur (Belgium), 2005.
6. T. Bolognesi and E. Brinksma, *"Introduction to the ISO Specification Language LOTOS"*, in Computer Networks and ISDN Systems 14:25-59, 1987.
7. C. Canal, E. Pimentel, and J.M. Troya, *"Compatibility and Inheritance in Software Architectures"*, in Science of Computer Programming 41:105-138, 2001.
8. X. Deng, M.B. Dwyer, J. Hatcliff, and M. Mizuno, *"Invariant-based Specification, Synthesis, and Verification of Synchronization in Concurrent Programs"*, in Proc. of the *24th Int. Conf. on Software Engineering (ICSE 2002)*, ACM press, pp. 442-452, Orlando (Florida), 2002.
9. C.A.R. Hoare, *"Communicating Sequential Processes"*, Prentice Hall, 1985.
10. P. Inverardi and S. Uchitel, *"Proving Deadlock Freedom in Component-Based Programming"*, in Proc. of the *4th Int. Conf. on Fundamental Approaches to Software Engineering (FASE 2001)*, LNCS 2029:60-75, Genova (Italy), 2001.
11. P. Inverardi, A.L. Wolf, and D. Yankelevich, *"Static Checking of System Behaviors Using Derived Component Assumptions"*, in ACM Trans. on Software Engineering and Methodology 9:239-272, 2000.
12. J. Magee and J. Kramer, *"Concurrency: State Models & Java Programs"*, Wiley, 1999.
13. R. Milner, *"Communication and Concurrency"*, Prentice Hall, 1989.
14. D.E. Perry and A.L. Wolf, *"Foundations for the Study of Software Architecture"*, in ACM SIGSOFT Software Engineering Notes 17:40-52, 1992.
15. M. Shaw and D. Garlan, *"Software Architecture: Perspectives on an Emerging Discipline"*, Prentice Hall, 1996.
16. T. Yavuz-Kahveci and T. Bultan, *"Specification, Verification, and Synthesis of Concurrency Control Components"*, in ACM SIGSOFT Software Engineering Notes 27:169-179, 2002.

A Appendix

We proceed by showing for each of the five steps of the transformation considered in isolation that the labeled transition system underlying the input process algebraic description of the step is isomorphic to the labeled transition system underlying the output process algebraic description of the step.

Step 1 (Rewriting Complex Choices). Denoted by M_1 the output of step 1, M_1 is obviously isomorphic to M because step 1 simply expands abbreviated choices or flattens nested choices.

Step 2 (Splitting the Equations). Denoted by M_2 the output of step 2, M_2 is isomorphic to M_1 because step 2 simply splits those equation branches that contain an interaction not occurring at the beginning of the branch itself. More precisely, if the branch of an equation E is of the form:

$$\beta_1 \rightarrow \beta_2 \rightarrow \ldots \rightarrow \beta_n \rightarrow \alpha \rightarrow P$$

where β_k is a local action for each $k = 1, \ldots, n$ and α is an interaction, then the following new equation is defined:

$$SPLIT\_1\_E = \alpha \rightarrow P$$

and the branch is rewritten into:

$$\beta_1 \rightarrow \beta_2 \rightarrow \ldots \rightarrow \beta_n \rightarrow SPLIT\_1\_E$$

The rewritten branch and the new equation considered as a whole are clearly isomorphic to the original branch. Similar is the case in which there is a choice among a set of interactions instead of a single interaction α.

Steps 3 and 4 (Building the Interacting Choice Equation and Rewriting Non-Local Equations into Setting Equations). The outcome of step 3 is the following interacting choice equation:

$$\begin{aligned}
&INTER_CH_EQ[eq : 0..n-1][g_\alpha_0 : Boolean] \ldots [g_\alpha_{m-1} : Boolean] = \\
&\quad (when((eq == 0)\ \&\&\ g_\alpha_{0,1}\ \&\&\ c_{0,1}) \quad \alpha_{0,1} \rightarrow P_{0,1}, \\
&\quad \ldots, \\
&\quad |when((eq == i)\ \&\&\ g_\alpha_{i,j}\ \&\&\ c_{i,j}) \quad \alpha_{i,j} \rightarrow P_{i,j}, \\
&\quad \ldots, \\
&\quad |when((eq == n-1)\ \&\&\ g_\alpha_{n-1,b_{n-1}}\ \&\&\ c_{n-1,b_{n-1}}) \quad \alpha_{n-1,b_{n-1}} \rightarrow P_{n-1,b_{n-1}} \\
&\quad)
\end{aligned}$$

where:

- n is the number of non-local equations of M_2.
- b_i is the number of interacting branches of non-local equation i ($0 \leq i \leq n-1$).
- m is the number of interactions occurring in the non-local equations of M_2.
- $\alpha_{i,j} \in \{\alpha_0, \ldots, \alpha_{m-1}\}$ is the interaction occurring at the beginning of branch j ($0 \leq j \leq b_i - 1$) of non-local equation i ($0 \leq i \leq n-1$).
- $c_{i,j}$ is the logical condition possibly inherited from branch j ($0 \leq j \leq b_i - 1$) of non-local equation i ($0 \leq i \leq n-1$).

In step 4 each non-local equation of M_2 is rewritten into a corresponding setting equation, in such a way that all possible local branches are preserved, while its branches that have been moved in step 3 to the interacting choice equation are replaced by a single invocation of the latter with suitably set actual values for parameters eq and $g\_$. More precisely, non-local equation i $(0 \leq i \leq n-1)$ is rewritten into the following setting equation:

$SETTING\_EQ_i =$
$(when(\bigvee_{j=0}^{b_i-1} c_{i,j})$
$\quad INTER\_CH\_EQ[i][\bigvee_{j\in\{0..b_i-1|\alpha_{i,j}=\alpha_0\}} c_{i,j}] \ldots [\bigvee_{j\in\{0..b_i-1|\alpha_{i,j}=\alpha_{m-1}\}} c_{i,j}],$
$|when(c_{i,b_i}) \quad \beta_{i,b_i} \rightarrow P_{i,b_i},$
$\ldots,$
$|when(c_{i,b_i+d_i-1}) \quad \beta_{i,b_i+d_i-1} \rightarrow P_{i,b_i+d_i-1}$
$)$

where:

- $\bigvee_{j=0}^{b_i-1} c_{i,j}$ is the logical disjunction of the conditions of the interacting branches of non-local equation i $(0 \leq i \leq n-1)$.
- $\bigvee_{j\in\{0..b_i-1|\alpha_{i,j}=\alpha_k\}} c_{i,j}$ is the logical disjunction of the conditions of the interacting branches starting with interaction α_k $(0 \leq k \leq m-1)$ in non-local equation i $(0 \leq i \leq n-1)$. This disjunction is taken to be false whenever the related index set is empty.
- d_i is the number of local branches of non-local equation i $(0 \leq i \leq n-1)$.

Note that constraint 3 guarantees the disjointness of the set of conditions associated with the interacting branches from the set of conditions associated with the local branches:

$$(\bigvee_{j=0}^{b_i-1} c_{i,j}) \wedge (\bigvee_{j=b_i}^{b_i+d_i-1} c_{i,j}) = false$$

The interacting choice equation and the group of setting equations considered as a whole are isomorphic to the group of non-local equations of M_2. Thus, the output $M_{3,4}$ of the steps 3 and 4 is isomorphic to M_2.

Step 5 (Rearranging the Interacting Choice Equation). The output M' of step 5 is isomorphic to $M_{3,4}$ because step 5 simply sorts the branches of the interacting choice equation and eliminates from it the redundancies introduced in the previous two steps.

More precisely, let us consider the three optimizations. The first optimization of the interacting choice equation can be done if an interaction α_k $(0 \leq k \leq m-1)$ occurs at the beginning of only one of the branches associated with a same value i $(0 \leq i \leq n-1)$ of eq. In fact, this means that the original non-local equation i (from M_2) contained only one branch j' $(0 \leq j' \leq b_i - 1)$ starting with α_k. Thus, referring to the setting equation i, it holds $\bigvee_{j\in\{0..b_i-1|\alpha_{i,j}=\alpha_k\}} c_{i,j} = c_{i,j'}$. Whenever the interacting choice equation is invoked by the setting equation i, $g\_\alpha_{i,j'}$ is set to $c_{i,j'}$ hence $g\_\alpha_{i,j'}$ && $c_{i,j'}$ coincides with $c_{i,j'}$ && $c_{i,j'}$. As a

consequence, condition $c_{i,j'}$ is redundant and can be removed from the related branch of the interacting choice equation.

The second optimization can be done if all the branches with the same initial interaction α_k are associated with a single value h ($0 \leq h \leq n-1$) of eq. In fact, whenever setting equation $i \neq h$ invokes the interacting choice equation, then $\bigvee_{j \in \{0..b_i-1 | \alpha_{i,j}=\alpha_k\}} c_{i,j} = \mathit{false}$ because the index set of the disjunction is empty. This means that $i \neq h$ implies $g\_\alpha_k = \mathit{false}$. Instead, when $i = h$ and $\alpha_{i,j} = \alpha_k$ for some $j = 0..b_i - 1$, we have that the check on eq is redundant because $(eq == i)$ && $g\_\alpha_{i,j}$ coincides with $g\_\alpha_{i,j}$. So, the check on eq can be removed from all the branches associated with interaction α_k.

The third optimization can be done if several branches are identical up to their boolean expressions. In fact, the two following branches:

$$|when((eq == i) \;\&\&\; g\_\alpha_{i,j} \;\&\&\; c_{i,j}) \quad \alpha_{i,j} \rightarrow P_{i,j},$$
$$|when((eq == h) \;\&\&\; g\_\alpha_{h,l} \;\&\&\; c_{h,l}) \quad \alpha_{h,l} \rightarrow P_{h,l}$$

where $\alpha_{i,j} = \alpha_{h,l} = \alpha_k$ and $P_{i,j} = P_{h,l} = P$, can be collapsed into the following single branch:

$$|when(((eq == i) \;||\; (eq == h)) \;\&\&\; g\_\alpha_k \;\&\&\; (c_{i,j} \;||\; c_{h,l})) \quad \alpha_k \rightarrow P$$

which is isomorphic to the two branches above considered as a whole. The same argument applies to an arbitrary number of branches that are identical up to their boolean expressions.