Communicating Process Architectures 2006
Peter Welch, Jon Kerridge, and Fred Barnes (Eds.)
IOS Press, 2006

Interacting Components

Bojan ORLIC and Jan F. BROENINK
CTIT and Control Engineering,
Faculty of EE-Math-CS, University of Twente
P.O.Box 217, 7500 AE Enschede, the Netherlands
{B.Orlic, J.F.Broenink}@utwente.nl

Abstract. SystemCSP is a graphical modeling language based on both CSP and concepts of component-based software development. The component framework of SystemCSP enables specification of both interaction scenarios and relative execution ordering among components. Specification and implementation of interaction among participating components is formalized via the notion of *interaction contract*. The used approach enables incremental design of execution diagrams by adding restrictions in different interaction diagrams throughout the process of system design. In this way all different diagrams are related into a single formally verifiable system. The concept of reusable formally verifiable interaction contracts is illustrated by designing set of design patterns for typical fault tolerance interaction scenarios.

Keywords. SystemCSP, CSP, components, contracts, contexts, fault tolerance, design patterns, formal methods, graphical modeling, simulation, hierarchical verification.

Preamble

This paper puts focus on the component related part of the SystemCSP design methodology. Although topics discussed here are presented in a self-sufficient way, full understanding of the paper might depend on familiarity with the SystemCSP graphical notation presented in the preceding paper [1] that puts focus on introducing visual elements of the SystemCSP methodology.

Introduction

Component-based software engineering is in practice most often based on the client-server architecture model, where one component (server) provides a certain service and the other component (client) uses that service. In some application areas, typical generic patterns are captured in the shape of standardized and precisely defined client side and server side interfaces (e.g. OPC [2]). This allows system integration based on components supplied by different vendors. Integration efforts are minimized as long as the used components adhere to these prescribed interfaces.

In a client-server system, the contract specifying interaction scenarios and adjustable parameters of service delivery is implicit and partially reflected in the interface definitions of the provided and required services. Sometimes those interfaces also offer services for contract parameter negotiation. Making an explicit entity that implements such a contract is considered to be unwanted overhead. This approach is justified for data processing systems

with clearly directed data flows starting with a client's request to the service provider, which can, in order to provide its service, further delegate part of its task to some other service provider(s) and in that way act as a client of the next component(s) in the client/server chain/tree. The obtained results travel in the opposite direction.

Complex client/server systems may however require existence of components that provide the management of interaction between several involved components. Components managing interaction of other components are in fact specifying and implementing explicit contracts governing interaction. For instance, a typical case would be providing, for fault tolerance reasons, redundancy in the form of replicated server components and an additional component/contract governing the interaction of the involved components.

Sometimes, e.g. in complex control applications, interaction between components is not natural to structure as a chain or tree of clients and servers. For instance, devices in a industrial production cell system need to cooperate as peers in order to provide a result. Every participating device has a precisely defined role, but it is not always clear what is the service, and if some component is in that interaction playing the role of a server or of a client. Instead, interaction between components is an interaction of peers that work together to achieve some higher-level behavior. In those situations, a structured approach is to introduce entities that will manage and supervise interactions between components. Such an entity is in fact defining an explicit contract between the involved components.

SystemCSP introduces *interaction contract* as a vehicle to manage interaction between components in structured and formally verifiable way. *Interaction diagram* depicts set of components centered around interaction contract. *Execution diagram* focuses on control flow elements that determine possible execution orderings of components. Control flow elements used in execution diagrams and binary relationships used in interaction diagrams are kept mutually consistent, allowing the same component to specify its execution relationship with other components in the interaction diagrams it participates in.

Section 2 of this paper provides information about related research efforts that were taken into consideration in designing the component framework of SystemCSP. Section 3 provides detailed information on the component framework introduced in SystemCSP. Section 4 describes some well-known fault tolerance design patterns in the shape of reusable interaction contracts, with a precise formally verifiable specification. At the end conclusions and recommendations for future work are presented.

1. Related Work

In [3], *Coordinating Atomic Actions* are introduced as a way to structure safety-critical systems involving complex concurrent activities. A coordinated atomic action (CA action) is an entity in which two or more threads of control implementing roles of the participating components meet and synchronize their activities performing atomically set of operations on a set of objects belonging to the CA action entity. In this way, a CA action behaves as a transaction and represents a general framework for dealing with faults and providing ways of recovery. Obviously, a CA action managing interaction contains more information needed for handling composite exceptional occurrences than any of the participating components in isolation. This makes CA actions a structured design pattern convenient for usage in safety-critical systems. The CA action design pattern is in [3] illustrated on a model of the Production Cell case study.

In [4], a *formal contract* is introduced as a design pattern that manages interaction among components in a side-effect-free way. A formal contract is defined as a state machine that codes interaction between components relying on a system of asynchronous modification requests from components to contract and state change notifications from

contract to components. The contract is promoted as an interaction entity that should substitute occam/CSP channels, since its ability to capture a complete N-directional specification of interactions between involved components makes it superior to channels usage. It is suggested that it is possible to transform a formal contract, being a state machine, into a CSP specification allowing in that way formal checking of interaction patterns managed by contracts. The usage of such a formal contract is foreseen as support useful during the full development cycle. The paper reports that usage of formal contracts in a real-life software problem resulted in a significant reduction of complexity and elimination of some typical problems related to unstructured use of concurrency.

WRIGHT [5] is an Architecture Description Language (ADL) that relies on CSP to describe the architecture of software systems. Basic abstractions of WRIGHT are: components, connectors and configurations. A component consists of two parts: computation and interface. The interface consists of ports. Each port is an interaction in which component can participate. The use of ports is to allow consistency checking and to guide programmers in the use of the associated component. The connector specifies the interaction between a set of components. It does that by providing the description of Roles representing expected behavior of participants and the *Glue* representing the specification on how the participating roles cooperate in the scope of the interaction managed by the connector. A *Configuration* is a set of component instances combined via connectors. WRIGHT is a textual way to describe architectures. No visual notation is provided.

In [6], making components contract aware is argued in order to be able to trust components employed in mission-critical applications. This paper deals with client-server architectures and identifies four levels of increasingly negotiable properties in the contract specification. On *basic (syntactic) level*, there is the interface description language (IDL) – like description of contract properties. This includes services/operations a component can provide/perform, associated input and output parameters and possible exceptions that may be raised during operation. Component frameworks that support (only) first-level contracts are for instance: CORBA, Component Object Model (COM), JavaBeans. Level 2 contracts are *behavioral contracts*. These contracts offer the possibility to specify pre-conditions, post-conditions and invariants for the performed operations. Typical examples of level 2 contracts are "design by contract" in the Eiffel language [7, 8], and Object Constraint Language (OCL) of UML. Level 3 are *synchronization contracts*. Contracts on this level specify behavior in terms of synchronizations and concurrency, e.g. whether a dependency between provided services is parallelism, sequence, shuffle, etc. The Service Object Synchronization (SOS) mechanism is used as an example for contracts on this level. Finally, level 4 contracts allow dynamic adaptation of the contract based on *Quality of Service* (QoS) requirements. TAO (the adaptive communication environment object request broker) is used as an example for a level 4 contract. Although the "four level" classification of contracts was introduced for implicit contracts of the client-server architecture, the classification is still a useful way to define more precisely the position of our notion of a contract.

2. Basic Concepts of the SystemCSP Component Framework

2.1 Components

In SystemCSP, besides a process describing the normal execution mode, components optionally contain a process managing possible reconfiguration scenarios and a process specifying the recovery activities upon occurrence of exceptional situations.

2.2 Interaction Contract

The interaction contract of SystemCSP is in fact the same concept as the connector concept in WRIGHT. The name interaction contract was chosen because it is more general and suits better its purpose then the name connector. Indeed, most simple interaction contracts (event, Any2One channels, buffered channels, etc) can be classified as connectors. But the entity specifying interaction among devices in an industrial production cell is more then just a connector.

Compared to formal contracts of Boosten, interaction contracts are directly implemented as CSP processes and there is no need for transformation in order to achieve formal checking. In addition, interaction contracts are not considered a substitute to channels, but a higher-level primitive described via event(channel)-based interactions with/among participating components.

The CA actions safety pattern [3] illustrates the importance of a centralized entity maintaining interaction between participating components and thus serves as a motivation for introducing interaction contracts as separate, explicitly existing entities. Compared to CA actions, interaction contracts are considered to be a more structured approach because they achieve the same purpose, but rely on a safer and more structured way to use concurrency. As in CA actions, one of the main powers of interaction contracts is the opportunity to nest handling of exceptional situations in contract facilities, where more knowledge is available about the current state of interaction than in participating components in isolation.

An *interaction contract* is an abstract entity whose main purpose is specifying and managing interactions between components. By defining interaction as an abstract entity (that can be instantiated in the same way as components can), a possibility for reuse of the design patterns captured in a form of interaction contracts is introduced. An interaction contract prescribes roles of the participants and offers additional interaction management support. It can introduce additional constraints in the way component instances interact, provide buffering support and exception handling facilities. A contract normally consist of three phases: checking preconditions, performing action and checking postconditions. An action can contain an interaction pattern specified via events or via subcontracts.

In the light of the four level of contracts classified in [6], interaction contracts provide natural support for the first three levels and the possibility to build an application specific Quality of Service layer on top of the first three layers. On the basic contract level, an interaction contract is described via event/channel interconnections, operations/actions they represent with associated input/output parameters and a defined set of possible exceptions that can propagate via the event/channel infrastructure. The second level is achieved by dividing every contract into three parts: optional checking of preconditions, the mandatory action and optional checking of postconditions. The third level is naturally supported by the CSP structure of the contract including the participating roles and the optional contract manager. In addition to the used channel/event ports, the CSP description of the contract encapsulates all possible scenarios for contract execution. Level 4 contracts can be built as an additional layer in the application specific contracts. General design patterns can be made to construct reusable QoS contract layers.

An interaction contract specifies roles for which a component willing to participate must provide an implementation. The implementation of a role must be a (trace) refinement in the CSP sense of the role description required in the contract. In CSP, the implementation is considered to be the (trace) refinement of the specification when a set of event traces that can be produced via execution of the implementation process is a subset of the set of traces that can be produced by the execution of the specification process. In other

words, a behavior of an implementation must stay within the behavior defined in the specification. This approach allows step-wise refinement during the design process and formal verification of even early stages of the design. The same role/component can be represented via several process descriptions on different abstraction levels ranging from a high-level specification to a low-level implementation. The refinement property between process descriptions on different levels can be formally verified.

Section 3 illustrates the concept of an interaction contract by introducing several fault tolerance design patterns in the form of reusable SystemCSP interaction contracts.

2.3 Contexts

A component can contain subcomponents and contexts. Contexts are nested inside components. There are two kinds of contexts: physical contexts and virtual contexts (e.g. a mind context or the internet). Via passages, new contexts can be opened while remaining in the previous contexts or a component can move from an old context into a new one (alike to following an internet link in new window or in existing one). For instance, a human-like component can at the same time be in at most one physical context, and in any number of virtual contexts. Self-aware components (e.g. humans or robots) are upon entering some context faced with a choice of interaction contracts offered by the context.

Interaction contracts are abstract definitions. Instances of interaction contracts (in further text, the name interaction contract is used) are located inside contexts. *Contexts* provide concrete environments in which interaction contracts can appear. A contract is abstract and when an instance of it is mapped onto some context, its notions needs to be mapped to objects in the context. For instance, a football game can be considered an interaction contract with certain rules. The notions used in the description of rules (goals, terrain, etc) must be mapped to existing objects in the context.

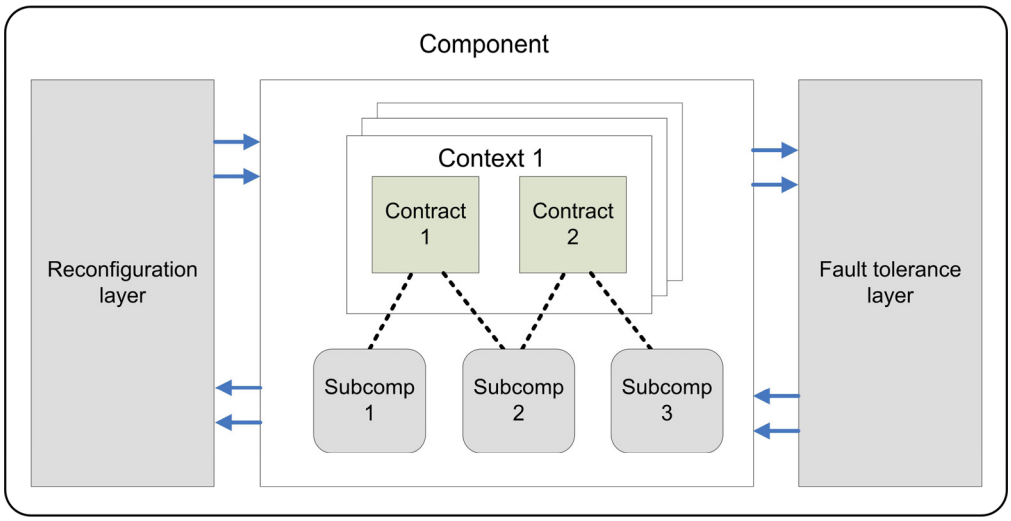

Figure 1. Contexts and contracts

2.4 Interaction and Execution Diagrams

The description of every component contains one *execution diagram* and one or more *interaction diagrams*. The *execution diagram* specifies the control flow, which determines the possible orderings in which subcomponents are executed. The *interaction diagram* focuses on interaction among subcomponents and contains a set of components grouped around interaction contract.

SystemCSP aims to allow the same component to participate in several different *interaction diagrams*. This is in a way similar to UML diagrams where one can focus on certain aspects of some entity in one diagram and on other aspects in other diagrams. Unlike in UML notation, where there is no relation between different diagrams, in SystemCSP all interaction views inside one component provide a single, consistent, formally verifiable, model of the system. This model is reflected in the execution diagram of the component.

In Figure 2, on the left-hand side, two interaction diagrams are given, and on the right-hand side the associated execution diagram is shown. Components B and C appear in both interaction diagrams because they engage in both interaction contracts. Component B is in the upper interaction diagram depicted via the black-box approach, and in the lower one via the transparent-box approach. Contract 1 from the upper interaction diagram is depicted via the transparent-box approach and Contract 2 from the lower interaction diagram is depicted via the black-box approach.

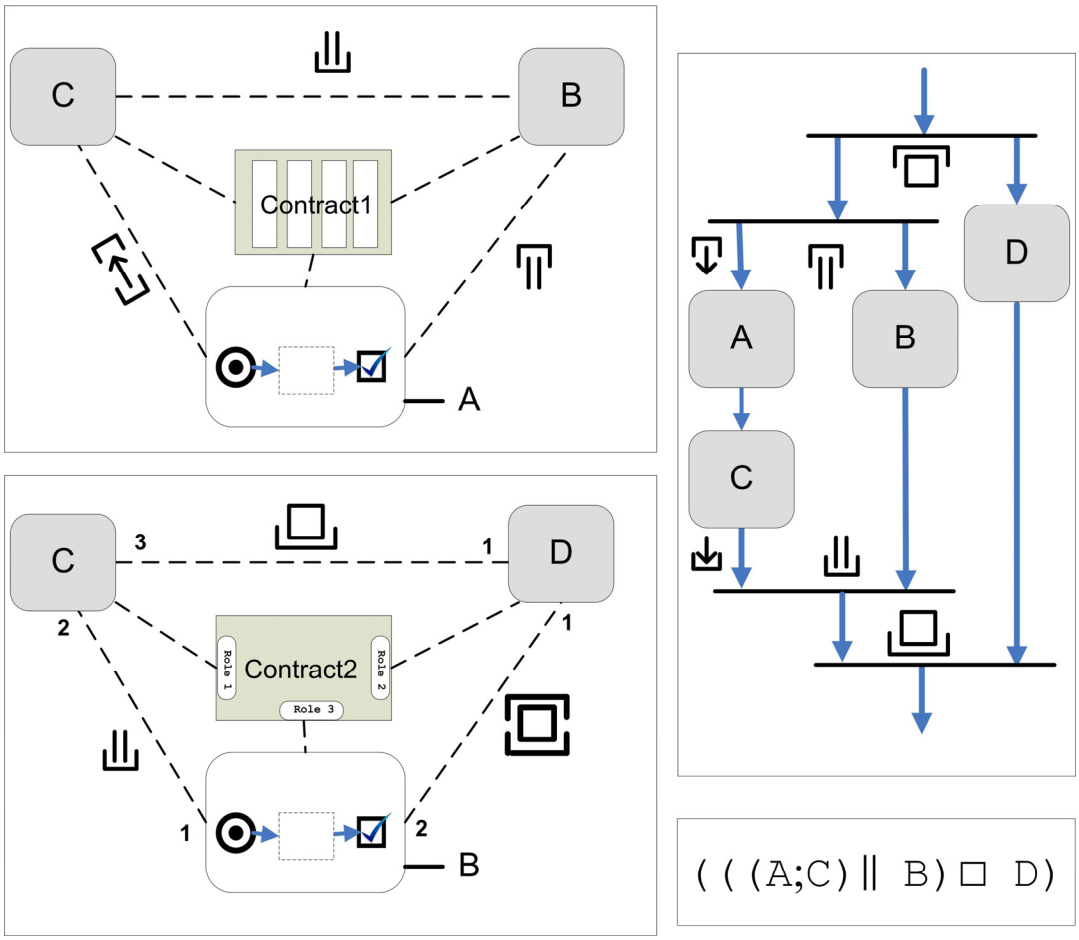

Figure 2. Specifying execution relationships in interaction views

Components participating in an interaction diagram do not exist in isolation, they are nested in some parent component that specifies a set of their possible execution orderings in the associated execution diagram (e.g. right-hand side diagram of Figure 2). Thus there is always some execution relationship between participating components. The binary relationships of GML [9, 10] served as an inspiration for introducing binary execution relationships between components in interaction views. The experience with using GML, showed that specifying binary relationships among components is very useful in early stages of the design, but is somewhat cluttering readability in later phases when focus is on control flow.

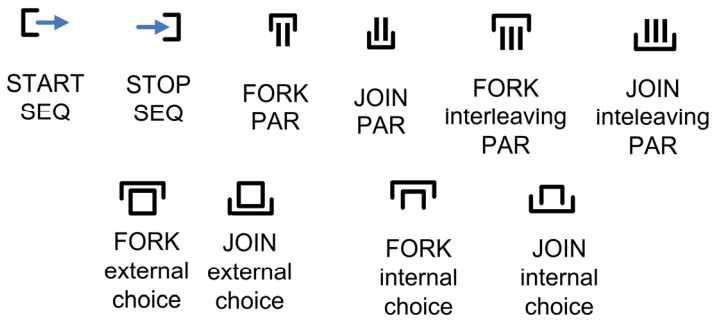

Figure 3. START and EXIT control flow elements

In the companion paper [1], a set of control flow elements is introduced (see Figure 3). The execution diagram on right-hand side of Figure 2 illustrates the usage of control flow elements to specify the execution pattern of a component whose internals are represented. Note that the START and EXIT elements can also be interpreted as binary relationships between any two related subprocesses.

Looking at components A and B in the execution diagram, one can notice that they are placed immediately after the FORK PAR control flow element, each one on start of one of the parallel branches. One can also interpret this as a FORK PAR binary relationship between components A and B. Thus, the relationship between A and B is stronger then PAR, because it implies that those two components are first in parallel branches and thus synchronizing on the START event. In the interaction diagram, dashed lines adorned with binary relationship symbols are used to specify a binary execution relationship between components. Components A and B are related in the upper interaction diagram with dashed line adorned with a FORK PAR symbol.

Components B and C are also in different branches, but instead of on START, they synchronize on a termination (EXIT) event. Thus, the binary relationship relating components B and C is of type JOIN PAR.

The FORK choice control flow element specifies that a choice is offered between component D and the parallel composition starting with components A and B. Thus, one can say that there is a FORK external choice binary relationship between components D and A and also between components D and B. Actually, components B and D are in addition also related via a JOIN external choice binary relationship.

When between two components both a FORK and a JOIN of the same kind of a binary relationship are present, we introduce one symbol instead of two and call such binary relationship STRONG relationship. The symbol for a STRONG relationship (see Figure 4), in addition to the symbol of the operator, contains both FORK and JOIN symbols. The relation between components B and D is thus a STRONG external choice.

Beside STRONG, WEAK binary execution relationships exist as well. A WEAK PAR exists between elements in parallel branches that do not synchronize on START or EXIT events. Components related via a WEAK PAR binary relationship can however synchronize on user-defined events. The WEAK interleaving PAR specifies that there is no synchronization *at all* between components belonging to parallel branches.

As it can be seen in Figure 4, besides PAR binary relationships, sequential and choice binary relationships also have WEAK and STRONG forms in addition to START and STOP forms. The sequential binary relationship in addition to its STRONG and WEAK form also has a PRECEDENCE form. It specifies that the involved components are executed one after another, but not necessarily immediately after each other.

WEAK and STRONG relationships are only implicitly specified in control flow diagrams, with STRONG ones represented via pairs of START and EXIT control flow elements and WEAK ones only as a relative position in the diagram. Still, execution diagram and a set of binary execution relationships specified in related interaction diagrams carry essentially the same information.

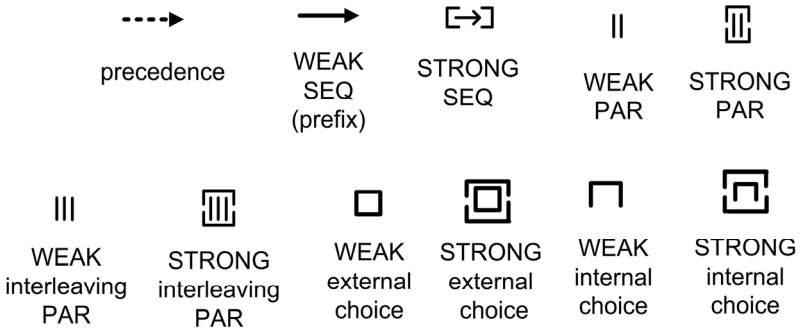

Figure 4. STRONG and WEAK relationships

Interaction diagrams can visualize an order of grouping (see numbers on the ends of relationships in the lower interaction diagram displayed in Figure 2) with lower number bearing the meaning of the closer execution control flow element. Note that in this ordering, the same number can be used only for exactly the same kind of JOIN/FORK contract.

By aligning elements of the execution diagram in such a way that the control flow goes downwards with FORKs and JOINs as horizontal lines connected via prefix arrows to the involved components below and above, a form that resembles the UML activity diagram is created (see the execution diagram in Figure 2).

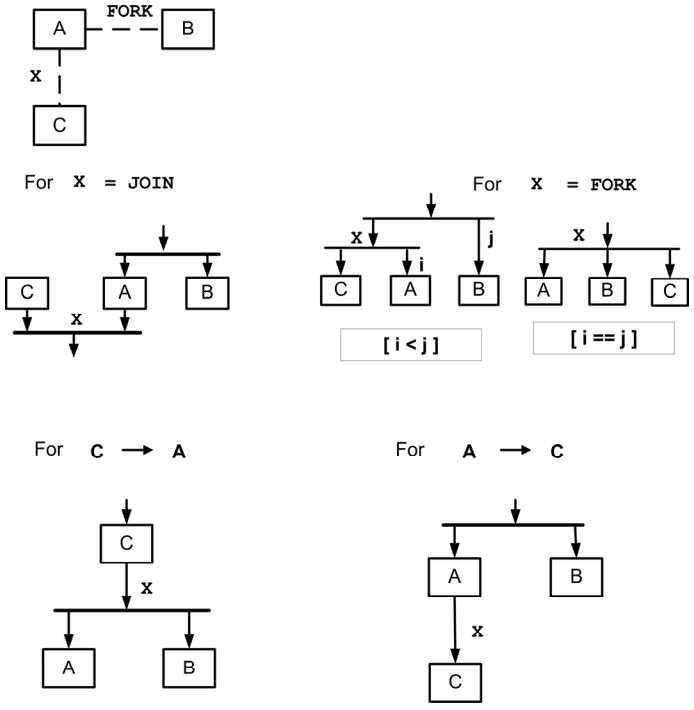

Figure 5. Transformation rules

In the general case, it is relatively straightforward to keep consistent execution relationships from views specifying interaction and execution. Components related via parallel or choice binary relationships in interaction diagram form separate branches in execution diagrams. Components related via binary sequential relationships are belonging to the same branch, with the one that is executed before located closer to the top of the branch. The intuitive set of rules for converting binary execution relationships between components into control flow elements of execution diagrams are illustrated for FORK kind of contracts in Figure 5. The set of rules for JOIN contracts is comparable.

In principle, the design starts with WEAK execution relationships (PRECEDENCE, WEAK SEQ, WEAK PAR, WEAK CHOICE) which are then either gradually refined to stronger variants (START, STOP and STRONG kinds of binary relationships) that specify implicit grouping or they stay weak if that is the intention.

An interaction diagram captures only the binary relationships relevant for the given interaction. Introducing an extended set of execution relationships with START (FORK), EXIT(JOIN), WEAK and STRONG forms, enables that one can specify a binary relationship between two components isolated from the rest of the environment. In practice, this allows that the same component can appear in different interaction diagrams in a way that is consistent across diagrams. Designs are in this way centered around interactions, with elements from different interaction diagrams strongly related. An execution diagram of a component is built incrementally, throughout the process of design, by adding restrictions in different interaction diagrams. All different diagrams are related into single formally verifiable system.

2.5 System Level Simulation

SystemCSP targets design of applications that run on top of distributed computer platforms and that interact with the physical environment (plant in further text). Interaction between computing nodes takes place over network interconnections and interaction between the application and the plant takes place via I/O interfaces. In our approach, nodes and plants are captured as components, and networks and I/O interconnections as system-level interaction contracts.

Defining the concurrency skeleton of a complete system in SystemCSP, allows one to perform/manage system level co-simulation between different domains.

The intention to use SystemCSP for system-level specification, design, implementation and co-simulation of distributed control systems gives justification for naming the complete framework SystemCSP.

A similar approach was already tested in related work [11], where the system-level simulation framework for co-simulation of the complete distributed system was based on a occam-like approach. The focus point of that simulation framework was prediction of the influence of network delays on the behavior of embedded control systems. Execution times of code blocks were considered negligible compared to network delays and the network simulator was reused from the TrueTime [12] simulation framework.

2.6 Potential for Hierarchical Verification

Consistent usage of interaction contracts on all hierarchy levels in the developed system has a potential to enhance the possibility to perform hierarchical verification. This is the case when an interaction contract specifies one complete self-sufficient interaction pattern that does not synchronize with the rest of the system. As such it can be formally verified as a separate unit in isolation from the rest of the system.

Formal verification of an interaction contract is in fact constructing equivalent state machine for the composed roles and the contract management layer. Such an equivalent state machine can for instance besides all possible traces, also capture the time properties.

A formally verified interaction contract can be substituted with a simplified interaction pattern relating involved components directly. For instance, in case when roles are composed via a STRONG parallel relation, it would suffice to replace, in all participating components, the complete role implementations with a single barrier synchronization point at those points where components engage in the roles required by the contract. If such simplification is systematically performed in bottom-up manner for all contracts and subcomponents inside some component, simplified equivalent state machines are obtained that represent the roles of this higher-level component. Applying this method consistently while progressing through the hierarchy of components in a bottom-up approach, it is possible to perform hierarchical verification, and in that way to decrease the significance of the state-explosion problem inherent in formal checking methods. This potential of interaction contract to serve as a vehicle for hierarchical bottom-up verification of systems is yet to be explored.

3. Fault Tolerance Design Patterns as Reusable Interaction Contracts

In this section, a set of fault tolerance design patterns useful for the development of real-time safety-critical distributed systems is presented. Patterns are designed and used to illustrate the usage of interaction contracts as reusable units in the practice of software development. The second aim of this section is to test the capabilities and expressiveness of SystemCSP when used as a vehicle in visualizing and structuring concurrency during the design of complex concurrent systems.

In fault tolerant systems, effort is made to design system that can continue providing required or degraded service despite the presence of faults in the system. A *fault* in a system can cause an *erroneous state* of some component. This error can further propagate and cause a *failure* of the expected service delivery. Faults can be *transient* and *permanent*. Fault tolerance can be seen as a process consisting of *error detection*, *error containment* (isolating error from spreading further), *error diagnosis* and *error recovery* [13].

In a SystemCSP-based system, functional error detection is naturally located in precondition and postcondition tests related to the execution of interaction contracts and subcontracts. Detecting errors in the timing relies on the *watchdog* design pattern. Upon detection of an error in an interaction contract, this contract can halt further progress of the interaction and in that way isolate the error from spreading further. An interaction contract is also a natural place to perform error diagnosis, since an interaction contract can possess more information about the current state of the interaction than the participating components on their own. The purpose of error recovery is to substitute an erroneous state with an error-free state. This state can be some previously saved state or its degraded part or it can be a new error-free state.

Forward error recovery attempts to handle errors by finding a new state from which the system can continue further operation. Usually it is based on *replication* redundancy. Replication can be done in software or in hardware (replicated specialized hardware or complete nodes or network interconnections). Forward error recovery is predictable in terms of time and memory overhead and thus often used in real-time systems [14].

Backward error recovery handles erroneous states by restoring some previous error-free state. Backward error recovery is especially suited for handling errors caused by transient faults. It has also the capability to handle unpredictable errors. The most widely used backward error recovery mechanism is *checkpointing* [14].

Another useful fault tolerance design pattern is *exception handling*. It can have *termination* or *resumption* semantics. The *take-over* operator of SystemCSP [1] covers the termination semantics. The resumption semantics upon occurrence of an exceptional situation is in our case just delegating exception handling to another part of the same process. Exceptions that cannot be handled are propagated across component boundaries. A special design pattern is introduced to perform this in a clean and formally verifiable way.

This section tries to introduce design patterns for some of the most important fault tolerance mechanisms: watchdog, replication, monitoring, event poisoning and checkpointing. First, the watchdog design pattern is introduced as a way to detect timing faults. Next design patterns that implement several different kinds of replica management are introduced. Monitoring as an important activity in safety-critical system is also introduced via a design pattern. The event poisoning design pattern is introduced as a way to transfer information about exceptional situations across component borders. At the end, a design pattern is introduced that uses checkpointing mechanism.

3.1 Watchdog Design Pattern

The interaction view specified in Figure 6 illustrates the interaction between a user-defined component and the timing subsystem component via the watchdog interaction contract. The watchdog contract is used to detect timing faults and to initiate built-in recovery mechanisms.

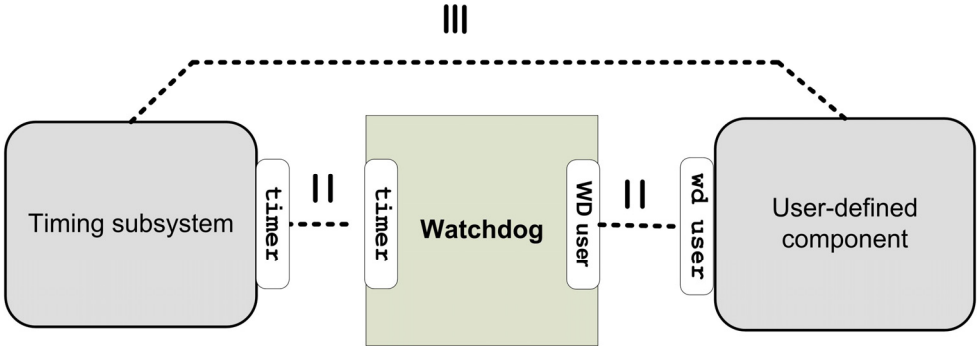

Figure 6. Interaction diagram: using a watchdog interaction contract

3.1.1 Timing Subsystem

Figure 7 introduces one possible design of a timing susbsystem. The purpose of this example is *not* to provide a ready-to-use design, but rather to illustrate how convenient SystemCSP is for making such designs. Besides, it introduces elements needed for understanding the working of the watchdog design pattern.

The timing subsystem contains several processes executed concurrently. HW_TIMER is a process implemented in hardware that produces instances of hardware interrupt processes (`HW_INT`) in regular intervals. The `HW_INT` process synchronizes with the CPU on event `int`, invoking in that way TIMER_ISR.

The process CPU acts as a gate that can disable (event `int_d`) / enable (event `int_e`) interrupts. When interrupts are enabled, event `int` can take place and as a consequence interrupt service routine TIMER_ISR will be invoked. In the case of an `int_d` event occurrence, the left branch in the guarded alternative of the CPU process is followed, which allows as a next event only the `int_e` event and thus the event `int` cannot be accepted, and as a consequence interrupts are not allowed.

TIMER_ISR increments the value of the variable `time` that it maintains. TIMER_ISR also maintains a sorted list of processes waiting on timeout events. Processes in this list, for which the time they wait for is less then or equal to the current time, will be awaken using the `wakeup` event. The awoken processes will be removed from the top of the list. In the case the awoken process is periodic, it is added again to a proper place in the waiting list.

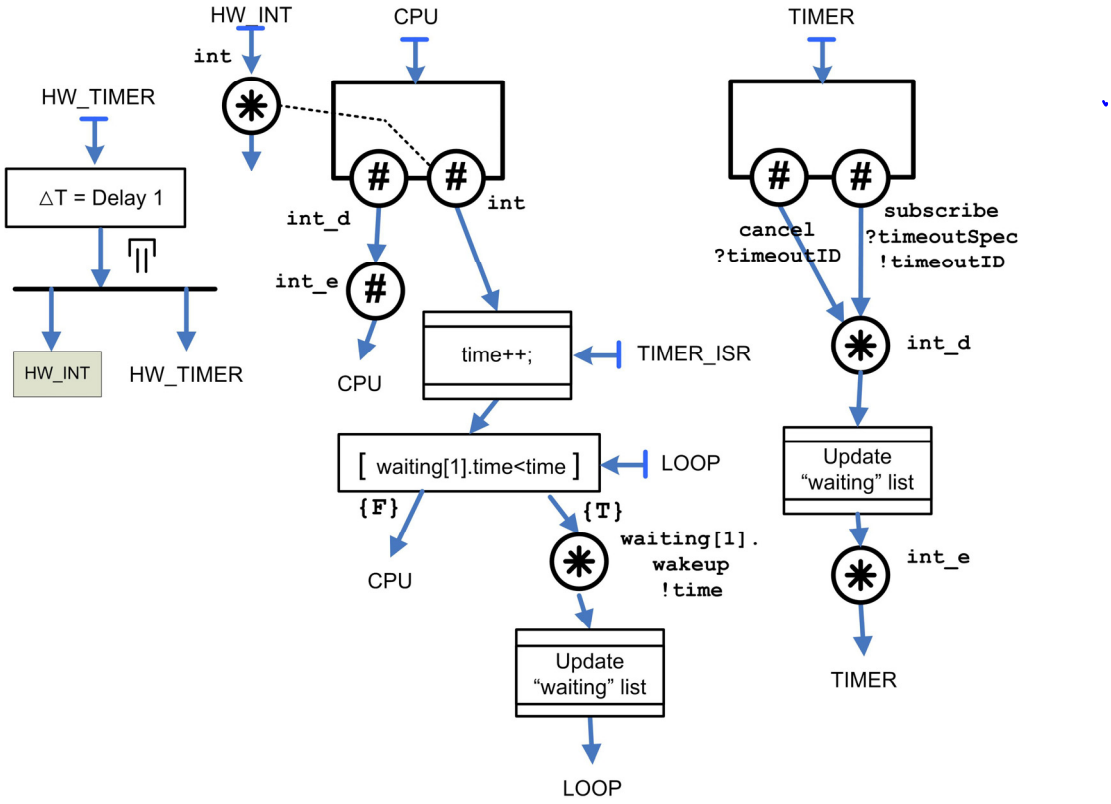

Figure 7. Timing susbsystem

Processes that are using services of the timing subsystem can, via the TIMER process, either subscribe (via event `subscribe`) to the timeout service or generate a `cancel` event to cancel a previously requested timeout service. Since these activities are actually updating the waiting list, this list must be protected from being updated in the same time by TIMER and TIMER_ISR processes. That is achieved in this case via disabling/enabling interrupts (`int_d` / `int_e` events).

3.1.2 Watchdog

The design pattern for the Watchdog process (see Figure 8) relies on services provided by the TIMER process.

A user of the watchdog contract must first initialize it by specifying timeout details in data communication related to the `start` event. Then the watchdog uses the `timer.subscribe` event to subscribe to receive the timeout service (one-shot or periodic depending on the mode parameter supplied by user) of the timing subsystem. After subscribing to the timeout service, the watchdog is ready to accept any of the three events – timeout from the TIMER_ISR process (`wakeup` event), or `hit` or `cancel` events initiated by the process guarded via this watchdog. In case when the user process initiates a

`cancel` event, the job of the watchdog is finished and it can cancel its timeout service and successfully terminate. Event `hit` will update a flag that keeps track of whether the `hit` event took place before timeout occurred. When a timeout occurs, the status flag will be checked and if the event `hit` did not take place, further execution of the guarded process is interrupted with the `abort` event. Depending on the mode, the watchdog will either prepare itself for the next iteration by resetting the status flag or it will cancel the timeout service and successfully terminate. Figure 8 captures watchdog interaction contract with definition of contract manager and the specification of the roles TIMER and WD USER.

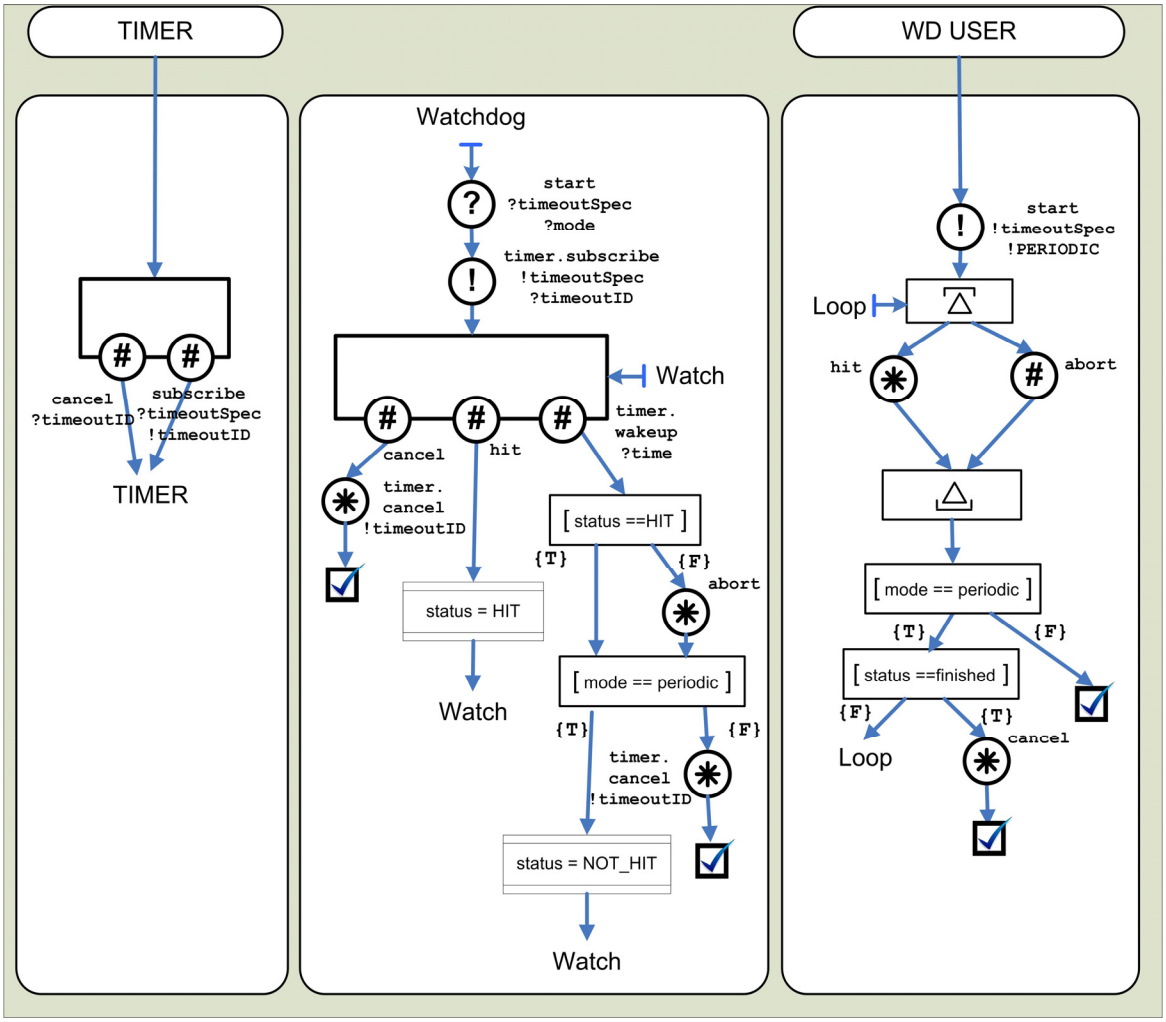

Figure 8. Watchdog interaction contract

3.1.3 *One-shot Watchdog Use*

The process Watchdog user (depicted in Figure 9) implements `wd_user` role specified in watchdog interaction contract given in Figure 8. While required role describes interaction that allows for both periodic and one-shot use, the implementation of the role given in Figure 9 uses a watchdog in a one-shot configuration. First, the watchdog is activated via the event `start`. The next part of program is guarded by the "take-over" operator [1] – SystemCSP equivalent of interrupt operator from CSP. In case when the watchdog signals a timeout by initiating the `abort` event, the normal execution branch is *taken-over* by the branch that handles the watchdog timeout. In the normal execution branch, after the normal execution is finished, the `hit` event is initiated to update the status of watchdog process.

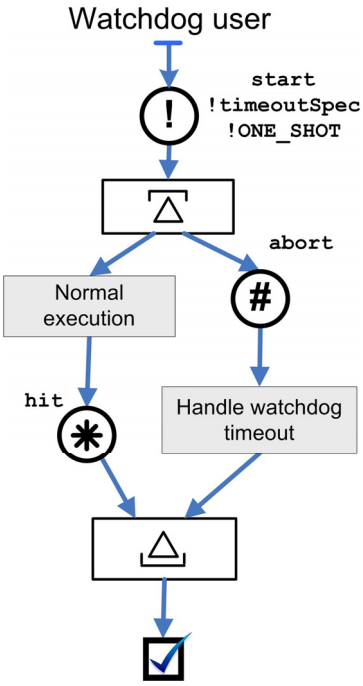

Figure 9. Watchdog user in one-shot configuration

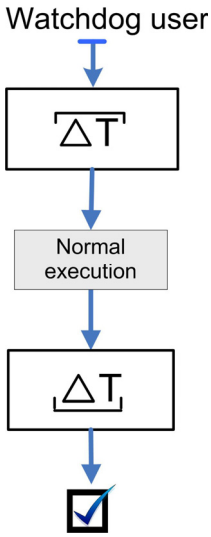

Figure 10. Symbol for using watchdog design pattern in one-shot configuration

Using the watchdog pattern is a useful safety design pattern, but it clutters the overview when one is interested in the normal execution flow only. For that reason, a special symbol, as depicted in Figure 10, is introduced as an abbreviation for the watchdog design pattern used in a one-shot configuration. The `start` and `hit` events and the way watchdog timeouts are handled, are considered to be part of the watchdog operator. Thus, they need not to be depicted when the normal execution flow is emphasized. Again, the watchdog operator is represented via one pair of FORK and JOIN control flow elements. The symbol used for watchdog operator is based on the combination of the *take-over* operator symbol and the letter T that implies timeout. This choice is made because watchdog usage is in fact using the *take-over* operator, where *take-over* can take place upon timeout events.

3.1.4 Periodic Watchdog Use

In a periodic usage of the watchdog, the difference is in the fact that that watchdog schedules a periodic timeout and that the guarded part of the user process is repeated in periodic iterations (compare Figure 9 and Figure 11). The assumption is that the process block `normal execution` (see Figure 11) starts with waiting on the periodic time event. The symbol used to abbreviate the design pattern with periodic use of the watchdog design pattern is depicted in Figure 12.

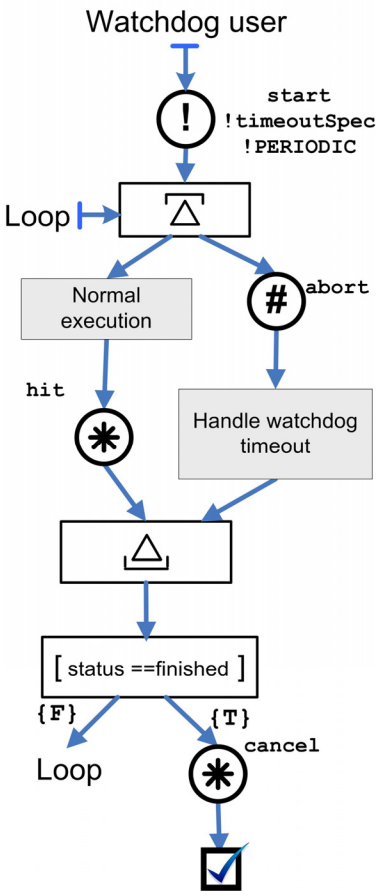

Figure 11. Watchdog user in periodic configuration

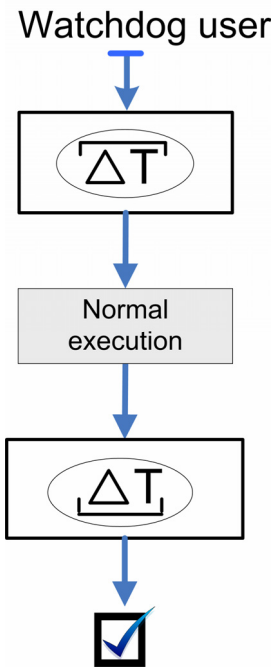

Figure 12. Symbol for using watchdog design pattern in periodic configuration

3.2 *Replica Management*

In Figure 13, an interaction diagram is displayed that relates a client component with a replicated server components via a replicaMgr contract.

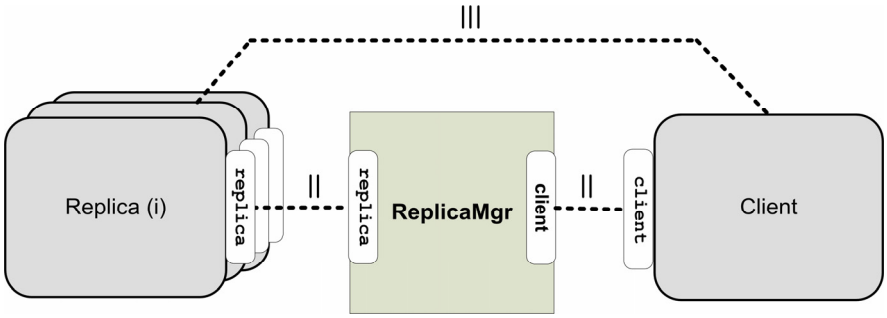

Figure 13. Replica management – interaction diagram

All code related to managing the replication is situated in the ReplicaMgr contract, which enables reusing the same components in different replication configurations. Replicas can be on same node or on different nodes, can be identical or based on different designs (N-version programming). The ReplicaMgr can be on the same node as some of the replicas or on a separate node. In this section, SystemCSP based designs are provided that specify “hot-standby”, “cold-standby” and “majority voting” types of ReplicaMgr.

3.2.1 "Hot Standby"

In this design pattern, upon receiving a request from a client, all replicas are activated, but only the result obtained from the fastest one is actually used. In the moment one of the replicas comes up with results, further execution of other replicas is aborted.

The Replica Mgr first receives a request from a client and then it will distribute the request in parallel to all involved replicas. In order to protect the interaction contract from the influence of failing components, sending the request to every replica is guarded using the watchdog design pattern. The replicas then work in parallel, and the interaction manager waits for a limited time (again the watchdog pattern is used) for one of the replicas to produce result. This kind of waiting is realized using an external choice element. In case one of the replicas comes up with the result, the other two replicas are aborted. Since an attempt to abort execution or to involve in any synchronization with an offline component can lead to a deadlock, the process of aborting those replicas is again guarded by a watchdog.

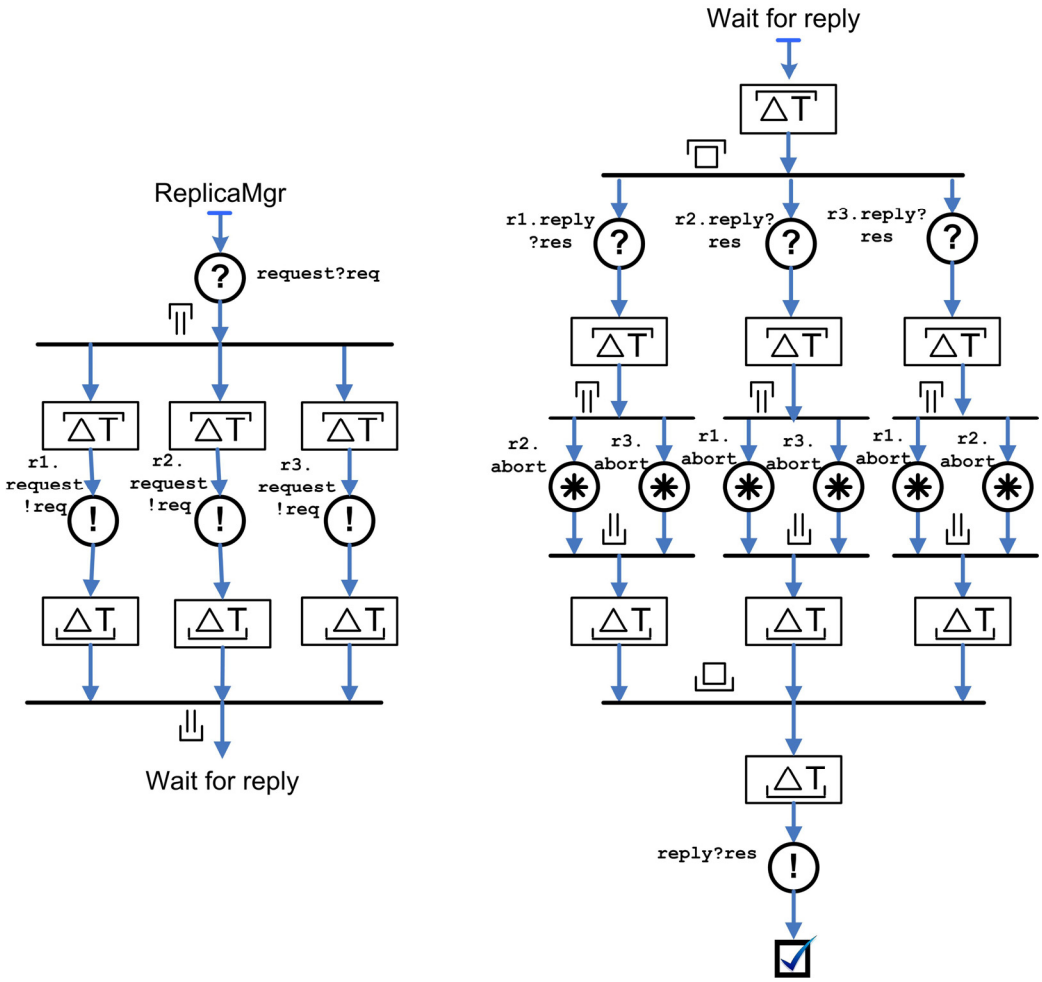

Figure 14. "Hot standby"

Note that in case when the replicas are invoked periodically and contain state (e.g. if replicas are controller implementations), the state of the replica that produced the result should be communicated together with result to the ReplicaMgr. In the next iteration, the state from the previous iteration should be communicated to all replicas. This approach will prevent internal states of replicas to drift away.

3.2.2 "Cold Standby"

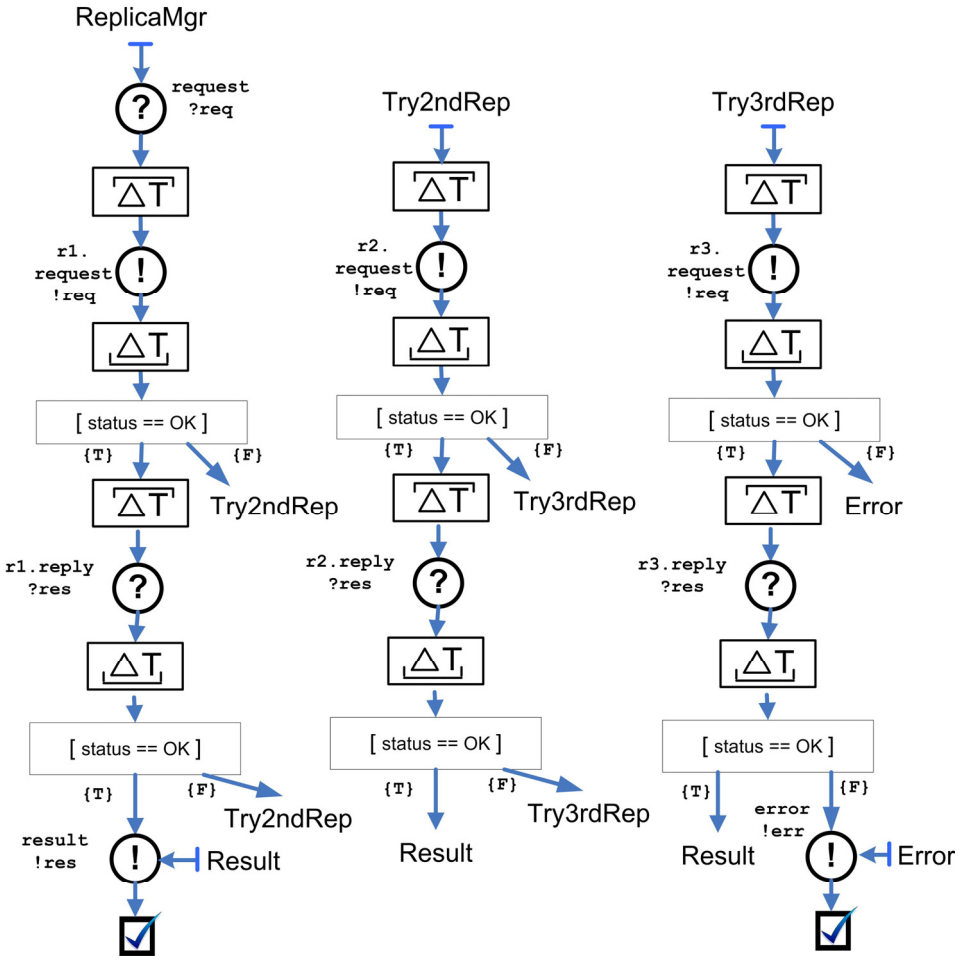

Figure 15. "Cold standby"

In the "Cold-standby" design pattern, a request is first forwarded to the first replica. If the first replica does not accept the request in the predefined time, then the request is forwarded to the second replica, and if it also fails to accept it, further on to the next replica in a chain.

After the request is accepted by one of the replicas, the ReplicaMgr waits for a reply for a predefined time interval. If the reply does not arrive, a request is sent to the next replica in the chain. If the replica replies, then the result is forwarded to the client. If no replica in the chain is able to provide the result, then the `error` event is initiated.

3.2.3 "Majority Voting"

In this design pattern, the request is sent in parallel to all replicas. The sequence of sending the request to the replicas and obtaining the reply from it, is guarded by the watchdog pattern. In that way, a failing replica cannot block the ReplicaMgr process. The obtained results and status flags are used in the "majority voting" process block to make an agreement about the correct result. In case when it is impossible to deduce a result, the `error` event is initiated.

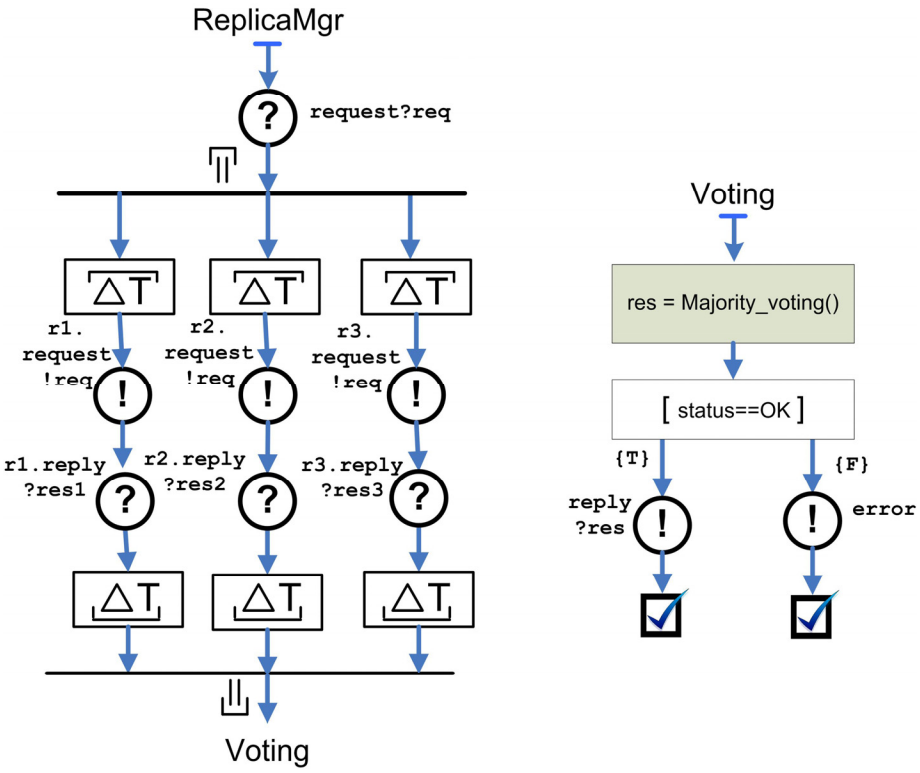

Figure 16. "Majority voting"

3.3 Monitoring

The monitoring design pattern enables safe monitoring of components and systems. Every process that needs to be monitored is associated with an EventLogger contract executing in Parallel with it. This EventLogger contract sends data further to the Monitor component. The Monitor component collects data from several monitored processes and can reason about different safety issues and invoke some safety measures if needed. Figure 17 displays the interaction diagram relating monitored process and monitor component via the EventLogger contract.

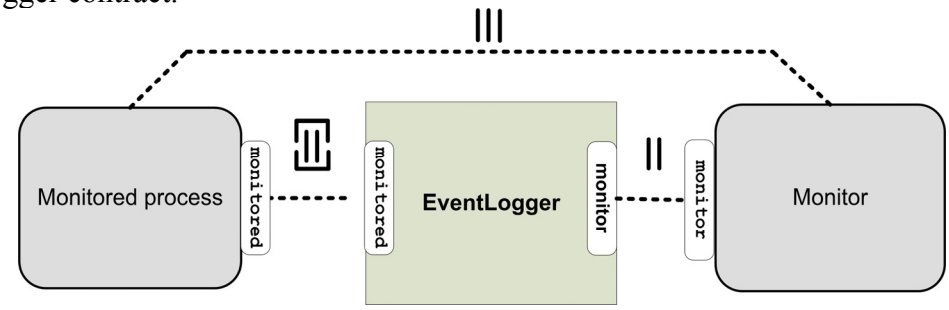

Figure 17. Monitor interaction pattern

In Figure 18, one can see that the monitored event is actually followed by an inserted additional event, which sends data to the EventLogger process. Note that monitored event can also be an internal dummy event, created only for monitoring purposes, and that in fact any value from the monitored process can be logged at a predetermined points in the process description via the EventLogger process. The EventLogger process logs data from the monitored process into a local buffer and upon the request transfers them further to the Monitor component. The Monitor component contains in shared memory the state of all variables relevant for its proper functioning.

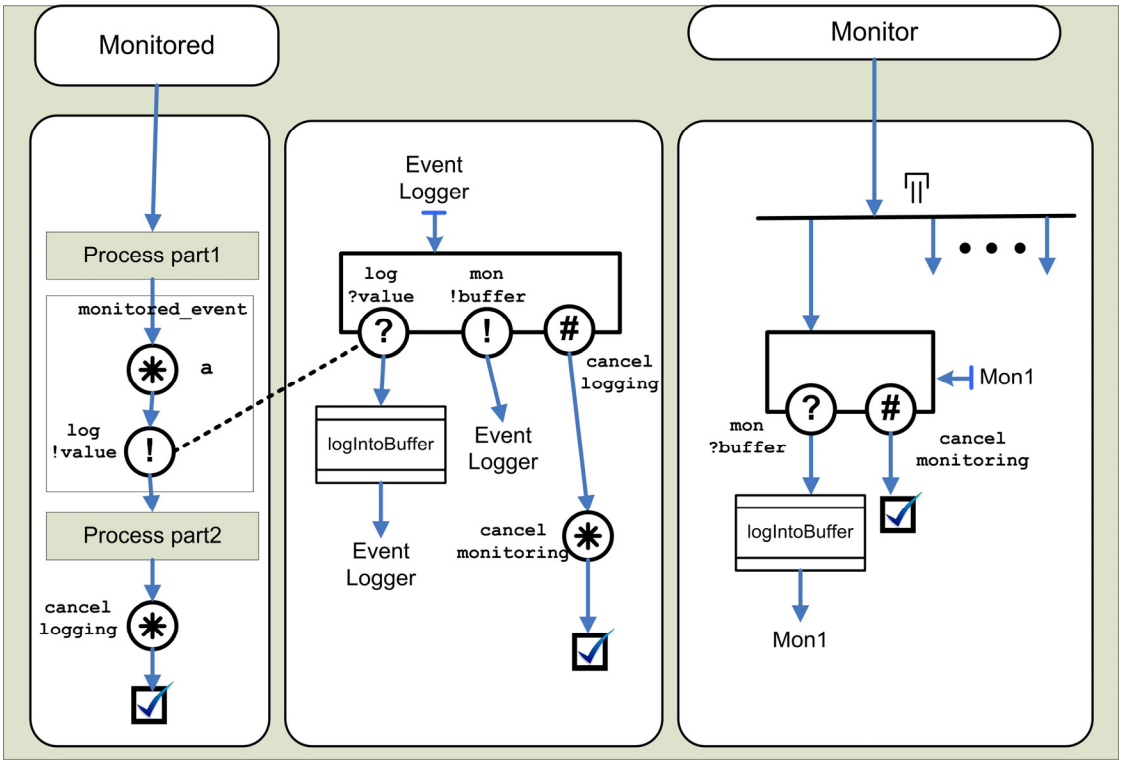

Figure 18. Details of monitoring interaction pattern

Figure 19 illustrates how the whole design pattern is abbreviated in designs in order to allow designers to focus on normal execution and hide away details of logging/monitoring. The symbol for the event that is monitored has rectangle around the monitored SyncEvent process. This gives an intuitive impression that it is a more complex process then just an EventSync process. The MON keyword is used to signify that the event is monitored. In addition, it is possible to specify the name of the monitor.

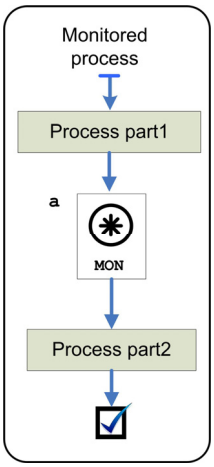

Figure 19. Monitoring symbol

3.4 Exception Handling

The termination model of exception handling (see the take-over operator in [1]) is not always convenient because it destructs all the work performed in the aborted process. Often, one wants to first attempt recovery and to avoid the need to abort the process. In SystemCSP, the recovery attempt upon the exceptional situation is still part of the same process. However, it is convenient to separate this part of the design into a special process block and allow in that way a visual separation of normal mode and exception handling mode (EHM). If an invalid state is observed in normal mode, further control flow may be designed to lead to the EHM via a recursion label.

Figure 20 depicts a case where EHMs of the contract and involved roles/components can interact and agree on the ways to handle the exceptional situation. The `normal mode` and `EHM` blocks are two visually separated parts of the same process. `Role1` of `Component1` offers both resumption and termination methods for exception handling. Resumption method relies on a jump to EHM part of the process that will attempt to handle an exception. The termination part is regulated by take-over operator and `Abort procedure` block. That is the reason that prefix arrows between those two blocks go in both directions. Contract named `Contract1` in this case, is a single process that is also visually divided into two process blocks: normal mode and EHM. The power of the configuration, where contract and component are structured in this way, comes from the fact that the interaction contract has additional knowledge about the current state of the interaction and can also obtain/maintain info on the status of the participating components. In that way, the interaction contract can pinpoint more precisely on possible causes of the exceptional situation and propose, to the participating components, ways to handle it.

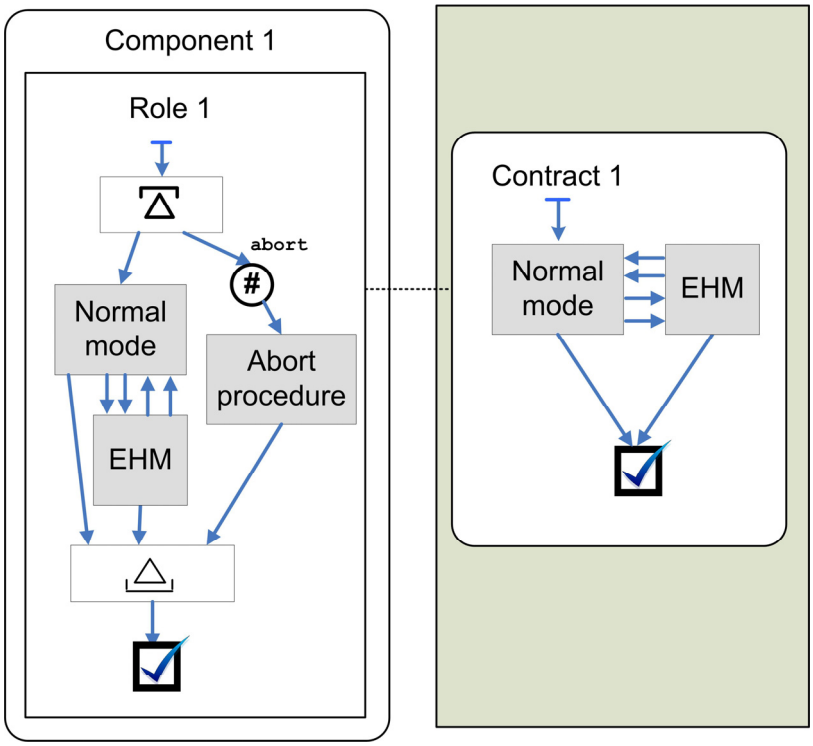

Figure 20. Cooperation of contract and component EHM layers

3.4.1 Event Poisoning

A convenient mechanism to pass information on the occurrence of an exceptional situation from a contract to the involved roles/components is to perform *event poisoning*. This concept origins from channel poisoning used for *graceful termination* in [15] and for transporting exception over process boundaries in the occam-like CT libraries [10, 16] [17].

Here, however, the mechanism is designed in a formally verifiable way. The events from the alphabet of some role participating in a contract that can produce exceptions are guarded for occurrence of exceptional situations. Guarding is performed by sending additional status information on every occurrence of the event that can be poisoned by an exception. In case an EHM of a contract needs to notify its participating role/component about an exceptional situation, it will initiate an event on which the component/role is waiting inside the interaction contract and send the exception info as a status. In normal mode of the role implementation, the obtained status is tested after every occurrence of the event guarded for exception and in case where an exceptional situation is detected, control flow is given to the EHM layer. From the EHM layer, after recovery, it is possible to return to the normal mode.

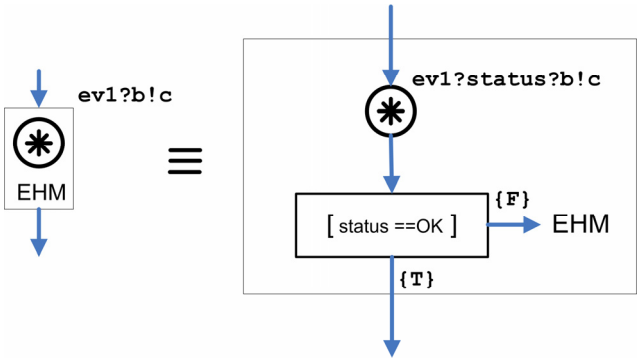

Figure 21. Event poisoning

To avoid cluttering designs with specifying testing of status info after every event occurrence on guarded EventSync processes, the special notation, as shown in Figure 21, is proposed for channels guarded for exceptional situations. The new symbol is a box around the eventSync process, with the keyword EHM written inside the box.

3.5 Checkpointing

Checkpointing is a backward recovery mechanism that relies on correcting an erroneous state by rolling back to some previous correct state. When interaction of several concurrently executing processes is guarded from faults in this way, a domino effect can occur. In such a domino effect one process causes another process to rollback, the other causes a third one to roll-back, and so on, which can result in rolling-back the first process even further and so on. This makes asynchronous checkpointing of interaction unsuitable for real-time systems where execution must be predictable in the sense of time and memory requirements. The proposed design pattern relies on interaction contract as a manager that keeps the roll-back process of involved components synchronized. In this way, the "domino-effect" is avoided and execution is predictable in the sense of time and memory requirements.

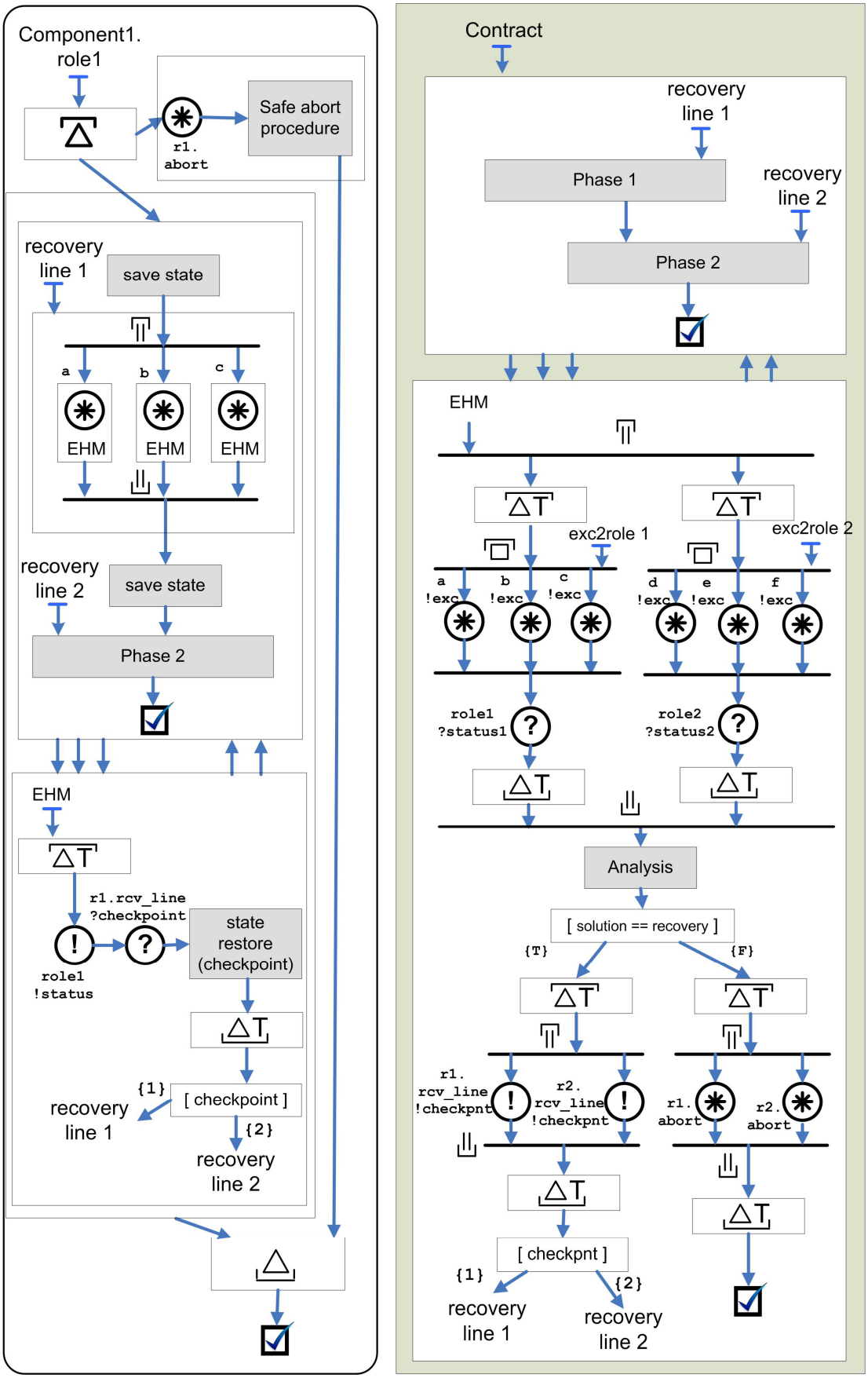

Figure 22. Checkpointing

In this design pattern, the *take-over* operator, *event poisoning, and* splitting a process into a normal part and an exception handling part are used as mechanisms to implement checkpointing. In example on Figure 22, two recovery lines are defined, splitting the participating components and contract into two phases. When inside Phase 1 or Phase 2 of the contract an exceptional situation is detected, control is given to the part of that process dedicated to handling exceptions. For every participating component, this part will offer an external choice on all of the EventSync processes belonging to the alphabet of that role. Instead of normal usage, these events will be used to transmit exception information to the components. Thus, the component that was blocked on one of those channels will be released from waiting and it will get a notification of failure of the attempted event. As a consequence, it will go into the part of its process definition that deals with exception handling. Then, it will use a dedicated channel to communicate its status to the contract or more precisely to the EHM part of the contract. The Contract will wait for a predefined period of time to obtain the status information of all involved components. After that, it will perform analysis and establish whether the interaction should be reverted to some recovery line or aborted. Its decision will be communicated to the participating components.

4. Conclusions

This paper introduces a component framework for the SystemCSP design methodology. Notion of reusable and formally verifiable interaction contracts as a way to manage interactions in a structured way is introduced. The concept is illustrated by designing reusable interaction contracts for the set of most commonly used fault tolerance design patterns.

The design patterns presented here illustrate that SystemCSP is a graphical notation that can provide intuitive and readable modeling in addition to the formal verification capabilities. The design patterns are concerned with often used, but rarely formalized fault-tolerance concepts. In that sense, since SystemCSP is directly translatable to CSP, this paper is also a contribution to formalizing those patterns.

Another important contribution of this paper is the introduction of a way to build systems around interaction diagrams, but still with a firm formally verifiable relationships preserved across diagrams. A particular component can participate in many different interaction diagrams where in addition to the managed interaction, execution relationships can be specified. This results in an incremental design of execution diagrams throughout the development process, by adding restrictions in different interaction diagrams. All different diagrams are related into single formally verifiable system.

In addition to future work specified in the companion paper [1], this paper gives design patterns that need to be implemented and tested in practice on the robotic setups in our lab.

References

[1] B. Orlic and J. F. Broenink, "SystemCSP - visual notation," presented at CPA, 2006, IOS Press, Amsterdam.
[2] "OPC, http://www.opcfoundation.org/."
[3] A. F. Zorzo, A. Romanovsky, J. Xu, B. Randell, R. J. Stroud, and I. S. Welch, "Using coordinated atomic actions to design safety-critical systems: a production cell case study," *Software: practice & experience*, vol. 29, pp. 677, 1999.
[4] M. Boosten, "Formal contracts: Enabling component composition," CPA 2003, IOS Press, Amsterdam.
[5] R. J. Allen, "A Formal Approach to Software Architecture," vol. PhD: Carnegie Mellon University, 1997.

[6] A. Beugnard, J-M. Jezequel, N. Plouzeau, and D.Watkins, "Making components contract aware," *IEEE Computer*, 1999.
[7] B. Meyer, " Applying "Design by Contract"," *Computer*, 1992.
[8] P. Nienaltowski, Meyer, B., "Contracts for concurrency," 2006.
[9] G. H. Hilderink, "Graphical modelling language for specifying concurrency based on CSP," *IEE proceedings. Software*, vol. 150, pp. 108, 2003.
[10] G. H. Hilderink, "Managing Complexity of Control Software through Concurrency," vol. PhD: University of Twente, 2005.
[11] M. ten Berge, B. Orlic, and J. F. Broenink, "Co-Simulation of Networked Embedded Control Systems, a CSP-like process-oriented approach," 2006.
[12] D. Henriksson and A. Cervin, "TrueTime 1.13-Reference Manual," Department of Automatic Control, Lund Institute of Technology, Lund, Technical report 2003.
[13] A. A. Avizienis, "Basic concepts and taxonomy of dependable and secure computing," *IEEE transactions on dependable and secure computing*, vol. 1, pp. 11, 2004.
[14] L. L. Pullum, *Software Fault Tolerance Techniques and Implementation*: Artech House, 2001.
[15] P. H. Welch, "Graceful termination --- graceful resetting," presented at 10th Occam User Group Technical Meeting, IOS Press, Amsterdam, 1989.
[16] D. S. Jovanovic, "Designing dependable process-oriented software, a CSP-based approach," vol. PhD: University of Twente, 2006.
[17] G. H. Hilderink, "Exception Handling Mechanism in Communicating Threads for Java," presented at CPA, 2005, IOS Press, Amsterdam.