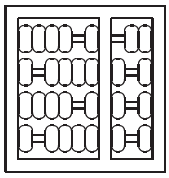

INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

Specification of the Idealised Fault-Tolerant C2 Component - Asynchronous Model

Paulo Guerra *Fernando Castor Filho*
Cecília Rubira

Technical Report - IC-04-06 - Relatório Técnico

June - 2004 - Junho

Specification of the Idealised Fault-Tolerant C2 Component - Asynchronous Model*

Paulo Guerra Fernando Castor Filho† Cecília Rubira‡

Abstract

The concept of idealised fault-tolerant C2 component(iC2C) was introduced as a means for constructing dependable component-based software systems in the C2 architectural style out of existing software components. It is derived from the concept of idealised fault-tolerant component, which aims at providing a structuring for systems which minimizes the contribution of the fault-tolerance mechanisms to their overall complexity. The use of iC2Cs as architectural blocks from which a system is built simplifies the task of building component-based, dependable systems. In this work we present informal and formal specifications for the iC2C. The formal specification is based on a state-machine view of the iC2C which emphasizes the functioning of its internal protocol and makes it easy to specify and prove properties over it. Furthermore, we make some considerations relevant to the implementation of the iC2C.

1 Introduction

Component-based software systems are used, nowadays, to solve a wide range of problems. According to Sprott, there is a general agreement in the industry that software components will bring profound changes in the way software is delivered[11]. In spite of this, the construction of dependable systems out of off-the-shelf components still represents a great challenge for software developers, since traditional software assurance technology has shown not to be effective for this kind of system[14]. Hence, alternative approaches have to be sought in order for obtaining trustworthy systems. One of these techniques is fault-tolerance, which is associated with the ability of a system to deliver services according to its specification in spite of the presence of faults.

For component-based software systems, fault-tolerance mechanisms must be introduced in the architectural level. This is dictated by the black-box view that component-based software development enforces. For dealing with the complexity associated with the task of introducing fault-tolerance mechanisms in component-based systems, Guerra introduced the concept of Idealised Fault-Tolerant C2 Component[4] (iC2C), which is an adaptation of the Idealised Fault-Tolerant Component[1] for the C2 architectural style[12].

*Correspondente address: Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP. Email: {asterio, fernando, cmrubira}@ic.unicamp.br

†Supported by FAPESP-Brazil, grant number 02/13996-2.

‡Partially supported by CNPq/Brazil, grant number 351592/97-0.

Software fault-tolerance in the architectural level is a relatively young research area. Most of the existing works emphasize the creation of fault-tolerance mechanisms, instead of the way they are incorporated into applications. The works of Cook and Dage[3] and Rakic and Medvidovic[9] present architectural level fault-tolerance mechanisms which bear great resemblance to some already existing software fault-tolerance techniques[2, 10]. Issarny and Banâtre[5] describe how to specify architectural exceptions and their respective handlers. In the C2 architectural style, support for the construction of dependable systems comes exclusively from the concept of Multi Version Connector[9], a mechanism created to allow the reliable upgrade of software components in a configuration.

In this work, we present the specification of the internal protocol of the iC2C. In this specification, the iC2C is modeled as a set of processes which interact through the exchange of synchronous and asynchronous messages. Our approach is formal: each process works as a finite state machine where state transitions are triggered by internal events or the receipt of messages. With this formal approach, properties may be proved over the internal protocol of the iC2C. Informal descriptions are also presented and accompany the specification of each process with the goal of making it easier to understand.

Since the main goal of this report is to serve as an implementation guide for the idealised C2 component, some implementation issues which are not directly addressed by the specification are also described. When adequate, solutions are presented and justified.

The report is organized as follows. Section 2 briefly introduces the C2 architectural style and the concepts of idealised fault-tolerant component and idealised fault-tolerant C2 component. Section 3 presents the specification of the iC2C, formally and informally. Section 4, discusses the implementation issues mentioned in last paragraph. Final conclusions are given in the last section.

2 Background

2.1 The C2 Architectural Style

The C2 architectural style first appeared in 1995[12] with the goal of increasing the reuse of components in the construction of component-based systems. At first, the C2 style was used to promote the reuse of graphical user interface components. Later experiments[7, 8] have shown that the C2 style is also adequate for the construction of diverse systems with heterogeneous features[7].

A C2 style architecture is composed by components, connectors, and connections. A component has an internal state, top and bottom domains and may have its own control thread. The bottom domain specifies the requests that a component is able to handle and the notifications it sends. The top domain specifies the requests the component issues, as well as the notifications it hopes to receive.

Components in a C2-style architecture communicate through asynchronous message-passing. The C2 style defines two types of message: requests and notifications. Notifications are sent downwards through an architecture, and requests are sent upwards. Notifications signal a change in the state of a component. In the C2 style a component should know nothing about the components in lower layers of the architecture[12]. Hence, the sending

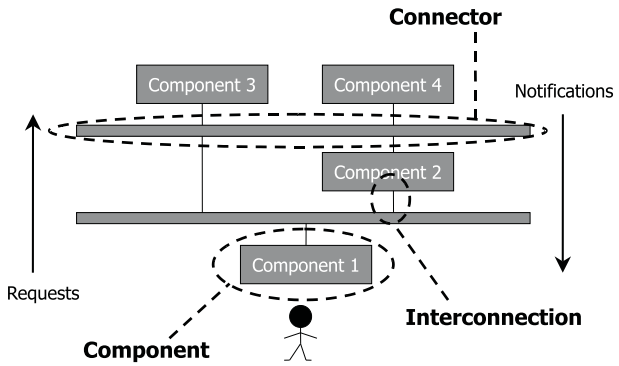

Figure 1: An example architecture presented in the C2 style.

of a notification does not involve any assumptions about which components are going to receive it. Requests ask components in upper layers of the architecture for some service to be provided.

It has already been mentioned that components communicate through message exchange. Components send messages through connectors, which connect components, as well as other connectors, and are responsible for the routing, filtering and broadcast of messages.

2.2 Idealised Fault-Tolerant Component

The activity of a system may be classified according to the kinds of answers it produces. When a system processes a request in the expected manner, without the occurrence of problems, the response it produces is called *normal*. Similarly, if the system is unable to process a request or does it so incorrectly, the response produced is called *abnormal*. The same can be said about the activity of a system. The activity of a system is *normal* when the system provides services according to its specification, as expected. When the system is unable to provide a service or provides it in an unexpected manner, its activity is *abnormal*. Usually, abnormal activity is related to the mechanisms needed to treat errors caused by diversions in the expected behavior of the system.

The concept of *idealised fault-tolerant component* (IFTC)[1] defines a component in which the parts responsible for the normal and abnormal activities are separated and well-defined. This structuring makes errors easier to detect and correct. The IFTC approach aims at providing an organization for systems which minimizes the contribution of fault-tolerance mechanisms to their overall complexity.

Figure 2 presents the internal structure of an idealised fault-tolerant component and the kinds of messages it sends and receives. The IFTC produces a *normal response* when a request is received and successfully processed. If it is a valid request but an error occurs while the request is being processed and the IFTC is not able to correct the error internally, a failure exception is raised. If the request is not valid, an interface exception is raised.

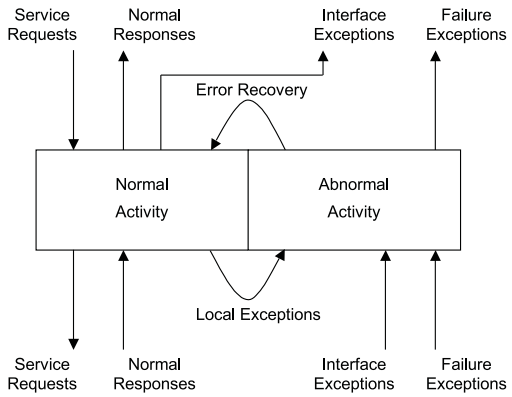

Figure 2: Idealised Fault-Tolerant Component[1].

2.3 Overview of the Idealised C2 Component

The idealised Fault-Tolerant C2 component[4](iC2C) is a concrete view of the concept of idealised fault-tolerant component. The work of Guerra, et al.[4], presents an architecture for the construction of C2 applications using the iC2C abstraction.

The iC2C serialises the processing of service requests. If two requests A and B arrive, in this order, at the iC2C, request A is processed first and only after it is over will request B be processed. Notifications received by the iC2C are processed in a different manner: they are sent downwards the architecture only when a request is being attended by the component.

The iC2C serialises the processing of requests with the aim of creating an error confinement region[1] within the component. Error confinement regions do not allow errors to propagate and affect parts of the system state which were not, originally, related to its occurrence. The confinement of an error to a part of the system state makes it easier to evaluate the severity and extension of the error and to recover the system state.

The use of iC2Cs as architectural blocks from which a system is built simplifies the task of building component-based, dependable systems. In the following section, the elements by which the architecture of the idealised C2 component is composed are described with a greater level of detail.

3 Specification of the iC2C

The structure of the iC2C is presented in Figure 3. The *NormalActivity* component implements the basic functionality of the iC2C, as well as error detection mechanisms. The *AbnormalActivity* component is responsible for error recovery and failure exception throwing. The iC2C_bottom, iC2C_internal and iC2C_top connectors control and direct the message flow inside the iC2C.

In this section, we present informal and formal specifications for the iC2C. A formal specification is necessary because, in spite of being easy to understand, an informal descrip-

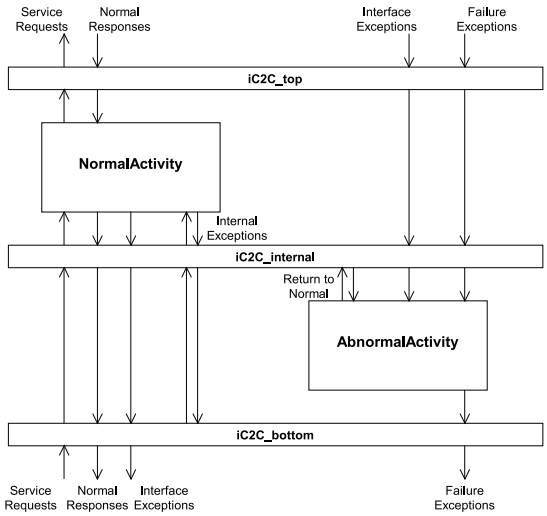

Figure 3: Internal organization of an idealised C2 component.

tion does not allow properties to be proved on the architecture of the iC2C. Since the iC2C is intended to be used in the construction of dependable systems, however, it is important to show that some properties are always true, independently of the application domain.

To achieve this task, the architecture of the iC2C has been modeled as a system composed of independent processes, each one of them functioning as a state machine. These state machines are described as *timed-automata* extended with data variables[6]. This formalism has been chosen because it has been employed in the modeling of many real-life protocols and concurrent systems[6] and due to the availability of free tools for the verification and validation of models which employ it.

The verification of properties on the model built will be done with the help of the *Uppaal* environment[6]. Uppaal is a toolbox for the modeling, simulation and model-checking of real time systems developed jointly by Uppsala University and Aalborg University. It is adequate for systems which may be modeled as a collection of non-deterministic processes with finite control structures. The properties we aim at proving, with the help of Uppaal, are:

- the iC2C never deadlocks;
- when the iC2C accepts a new request from a client, all of its sub-components are in their initial states;
- when the activity of the NormalActivity component normal, the AbnormalActivity component is idle.;
- while the AbnormalActivity component is handling an exception, the activity of the

NormalActivity component is not normal;

- while in normal activity, the iC2C_top connector is active with normal behavior;
- while recovering from an exception, the iC2C_top connector is active with abnormal behavior.

These properties have been proved using Upaal. In order to avoid state-space explosion, we have made some simplifications to their specifications. These are mostly restrictions to queue sizes and do not affect their validity in general cases.

3.1 iC2C_bottom Connector

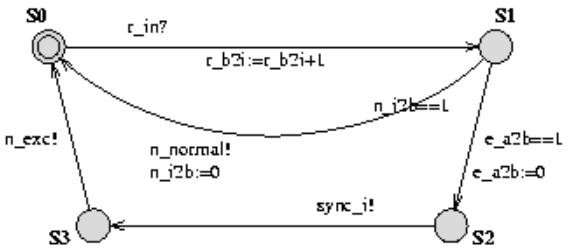

Figure 4: Timed automaton for the iC2C_bottom connector.

The main responsibility of the iC2C_bottom connector is to enqueue the requests received by the iC2C and guarantee that requests are always processed in a serialised manner. Other functions might include the generation of "heartbeat" messages, to avoid notifications of waiting indefinitely in the queue of the port of the top domain of the iC2C_top connector, and the unpacking of notifications, when many of them are sent through a single message.

Functioning of the iC2C_bottom connector: initially, the iC2C_bottom connector is idle. Requests arriving at the port of its bottom domain are enqueued. When no requests are being processed by the iC2C, it takes the first request out of the queue and sends it to the iC2C_internal connector. Once the request is being processed, the iC2C_bottom simply waits until a response is received. The processing of a request may produce two kinds of response: "normal" and "exceptional". Both are signaled by a notification received by the iC2C_bottom. In the first case, the notification is simply sent downwards the architecture.

If a notification received signals an exception, the iC2C_bottom connector must synchronize with the iC2C_internal connector before sending the notification downwards. This synchronization action aims at assuring that the iC2C returns to its initial state before the exception is sent. After the synchronization, the exception proceeds downwards the architecture.

State	Description
S0	iC2C is idle, waiting for a new service request.
S1	iC2C is processing a request. Waiting for its resulting (normal/exception) notification.
S2	The request generated an exception notification. Synchronizing with internal connector at its initial state.
S3	Synchronized with internal connector at its initial state after an exception result. The iC2C is ready to return to idle state.

Transition		Specification			Description
From	To	Guard	Sync	Assign	
S0	S1		r_in?	r_b2i:=r_b2i+1	A client request is taken from the input queue, at the iC2C bottom port, and sent up, to the internal connector
S1	S0	n_i2b==1	n_normal!	n_i2b:=0	A normal notification is received, from the internal connector, and is sent down, through the iC2C bottom port.
S1	S2	e_a2b==1		e_a2b:=0	An exception notification is received, from the AbnormalActivity component.
S2	S3		sync_i?		Synchronizes with the internal connector at its initial state.
S3	S0		n_exc!		An exception notification is sent down, through the iC2C bottom port.

3.2 iC2C_internal Connector

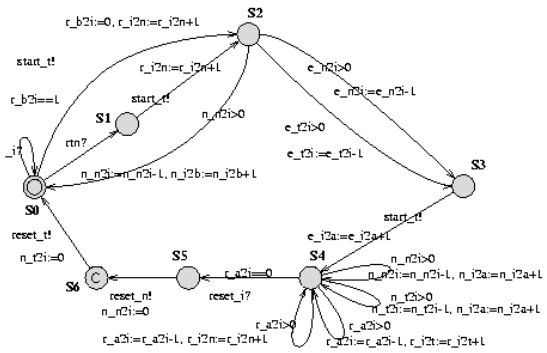

Figure 5: Timed automaton for the iC2C_internal connector.

Among the connectors of the iC2C, the iC2C_internal connector is the most complex. This connector is responsible for the routing of messages inside the iC2C. This task must be performed because the iC2C has two operation modes: *normal activity* and *abnormal activity*. The destination of the messages sent by the internal elements of the iC2C, whether or not they are restricted to the boundaries of the iC2C, or received by them depends on the operation mode the iC2C is in, in the moment the message is sent or received, respectively. For instance, if the iC2C is working in the abnormal activity mode, any notifications received must be delivered to the AbnormalActivity component, independently of their types.

Functioning of the iC2C_internal connector: initially, the iC2C_internal connector is idle. In the normal activity mode, requests received by the iC2C_internal are sent to the NormalActivity component, so that it can process them. When a normal notification is received by the iC2C_internal connector, it is relayed to the iC2C_bottom connector without additional processing.

The behavior of the iC2C_internal connector is different when it receives a notification which signals an exception. In this case, the iC2C_internal re-activates the iC2C_top connector (this connector automatically stops relaying notifications when an exceptional notification is received, as described in Section 3.3), sends an exceptional notification to the AbnormalActivity component, and enters the abnormal activity mode.

Once functioning in the abnormal activity mode, the iC2C_internal connector behaves differently. Notifications received by it, exceptional or normal, are relayed to the AbnormalActivity component. While handling an exception, the AbnormalActivity component may request services from the NormalActivity component or from other components located in upper layers of the architecture. The iC2C_internal connector delivers requests to the NormalActivity component and/or to the iC2C_top connector, depending on the type of the message.

When the AbnormalActivity component ends it error-recovery related activities (inde-

pendently of having succeeded or not), it sends a message requesting the iC2C_internal connector to return to its initial state. This request is only processed after all requests sent by the AbnormalActivity before it have been processed. When this condition holds, the iC2C_internal sends a request to the NormalActivity component asking it to also return to its initial state, discarding any notifications that might be enqueued in the port of its top domain. The iC2C_internal connector also sends a request to the iC2C_top connector, asking it to return to its initial state. These actions aim at preparing the iC2C to return to the normal activity mode.

State	Description
S0	Internal connector, normal activity component and top connector are idle (or stopped).
S1	Returning to normal operation after an internal exception.
S2	Normal processing of a request. AbnormalActivity component is idle.
S3	Normal processing was suspended after an exception. Normal activity component and top connector are stopped. AbnormalActivity may be started.
S4	Recovering from an exception.
S5	Error recovery is concluded. AbnormalActivity is stopped.
S6	Error recovery is concluded. NormalActivity and AbnormalActivity are stopped.

Transition		Specification			Description
From	To	Guard	Sync	Assign	
S0	S1		rtn?		A return-to-normal request is received, from the AbnormalActivity component.
S0	S2	r_b2i==1	start_t!	r_b2i:=0, r_i2n:=r_i2n+1	A client request is received, from the bottom connector and is sent up to the NormalActivity component, after activating the top connector.
S1	S2		start_t!	r_i2n:=r_i2n+1	A return-to-normal request is sent to the NormalActivity component.
S2	S0	n_n2i>0		n_n2i:=n_n2i-1, n_i2b:=n_i2b+1	A normal notification is received, from the NormalActivity component, and is sent down to the bottom connector.
S2	S3	e_n2i>0		e_n2i:=e_n2i-1	An (internal) exception notification is received, from the NormalActivity component.
S2	S3	e_t2i>0		e_t2i:=e_t2i-1	An (external) exception notification is received, from the top connector.
S3	S4		start_t!	e_i2a:=e_i2a+1	Re-activates the top connector and enter the abnormal state, sending an exception notification to the AbnormalActivity component.
S4	S4	n_n2i>0		n_n2i:=n_n2i-1, n_i2a:=n_i2a+1	A (normal / exceptional) notification is received from the NormalActivity component and is sent to the AbnormalActivity component.

S4	S4	n_t2i>0		n_t2i:=n_t2i-1, n_i2a:=n_i2a+1	A (normal / exceptional) notification is received from the top connector and is sent to the AbnormalActivity component.
S4	S4	r_a2i>0		r_a2i:=r_a2i-1, r_i2t:=r_i2t+1	A request is received from the AbnormalActivity component and is sent to the top connector.
S4	S4	r_a2i>0		r_a2i:=r_a2i-1, r_i2n:=r_i2n+1	A request is received from the AbnormalActivity component and is sent to the NormalActivity component.
S4	S5	r_a2i==0	reset_i?		A request to synchronize at the initial state is accepted. There should be no pending requests from the AbnormalActivity.
S5	S6		reset_n!	n_n2i:=0	Synchronizes with the NormalActivity component at its initial state preparing to return to normal operation, discarding any notifications from it that may be pending.
S6	S0		reset_t!	n_t2i:=0	Synchronizes with the top connector at its initial state preparing to return to normal operation, discarding any notifications from it that may be pending.
S6	S6		sync_i?		A request to synchronize at its initial state is accepted.

3.3 iC2C_top Connector

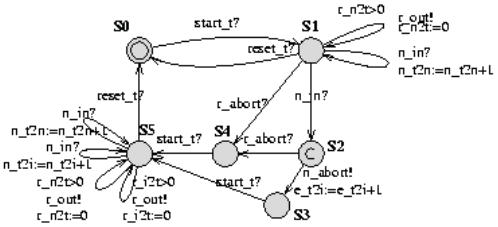

Figure 6: Timed automaton for the iC2C_top connector.

The iC2C_top connector enqueues in its top domain notifications received by the iC2C and does not allow these to be processed until the iC2C is attending a request. It then directs notifications received by the iC2C to the appropriate internal component, depending on the notification type(normal or exceptional) and the operation mode of the iC2C(normal or abnormal activity). Another task the iC2C_top is responsible for is the sending of service requests upwards the architecture.

The iC2C_top connector also works as a domain translator for messages. For example: in some contexts, an exceptional notification should be treated as a normal notification by an iC2C, that is, the error recovery mechanism of the iC2C should not be activated by its receipt. This behavior is achieved by the conversion of the exceptional notification to a normal one.

Functioning of the iC2C_top connector: initially, the iC2C_top connector is stopped. When stopped, it does not relay messages. Notifications received are enqueued in the port of its top domain. Requests do not need to be taken into consideration, due to the construction of the internal protocol of the iC2C. When the iC2C_internal connector receives a service request, it activates the iC2C_top.

The behavior of the iC2C_top connector changes when it is activated. Requests enqueued in the port of its bottom domain are sent upwards the architecture, so that they can be processed. Normal notifications enqueued in the port of the top domain are sent to the NormalActivity component. If no exceptional notifications are received by the iC2C_top, it works this way until the iC2C_internal sends it a request to return to its initial state and indicating that a request has been successfully processed.

When an exceptional notification is received, the iC2C_top interrupts its activities, sends a notification to the NormalActivity component asking it to do the same, and sends an exceptional notification to the iC2C_internal. Besides exceptional notifications coming from components located in upper layers of the architecture, it is also possible that the iC2C_top receives a request to interrupt its services from the NormalActivity component. This kind of request signals the occurrence of an internal exception.

As soon as the iC2C enters the abnormal activity mode, the iC2C_internal re-activates the iC2C_top, sending it a request. Once re-activated, the iC2C_top connector resumes its

activity, sending requests enqueued in the port of its bottom domain upwards. Notifications enqueued in the port of its top domain are sent to the NormalActivity component or to the iC2C_internal connector, depending on the message type. The iC2C_top behaves in this manner until it receives a request to return to its initial state from the iC2C_internal, signaling the end of the abnormal activity.

State	Description
S0	Stopped.
S1	Operating with normal behavior. Notifications are sent to NormalActivity component.
S2	An external exception was received. Synchronizing state with NormalActivity to suspend normal processing.
S3	Normal processing suspended. Sending external exception to internal connector.
S4	Normal processing was suspended by the NormalActivity component, after an internal exception.
S5	Operating with abnormal behavior (error recovering). Accept requests/send notifications from/to NormalActivity and AbnormalActivity components.

Transition		Specification			Description
From	To	Guard	Sync	Assign	
S0	S1		start_t?		A request to activate the connector is received.
S1	S0		reset_t?		A request to synchronize at its initial state is received.
S1	S1	r_n2t>0	r_out!	r_n2t:=0	Send forward pending requests received.
S1	S1		n_in?	n_t2n:=n_t2n+1	A notification is taken from the input queue, at iC2C top port, and is sent down to the NormalActivity component.
S1	S2		n_in?		An (exception) notification is taken from the input queue, at iC2C top port. Prepares to suspend normal processing.
S1	S4		r_abort?		A request to suspend normal processing is received, from the NormalActivity component.
S2	S3		n_abort!	e_t2i:=e_t2i+1	Synchronizes with the NormalActivity component to suspend normal processing and sends an exception notification to the internal connector.
S2	S4		r_abort?		A request to suspend normal processing is received, from the NormalActivity component.
S3	S5		start_t?		A request to (re)activate the connector is received.

S4	S5		start_t?		A request to (re)activate the connector is received.
S5	S5	r_i2t>0	r_out!	r_i2t:=0	Send forward pending requests received from the internal connector.
S5	S5	r_n2t>0	r_out!	r_n2t:=0	Send forward pending requests received from the Normal Activity component.
S5	S5		n_in?	n_t2n:=n_t2n+1	A notification is taken from the input queue, at iC2C top port, and is sent down to the NormalActivity component.
S5	S5		n_in?	n_t2n:=n_t2n+1	A notification is taken from the input queue, at iC2C top port, and is sent down to the internal connector.
S5	S0		reset_t?		A request to synchronize at its initial state is received.

3.4 NormalActivity Component

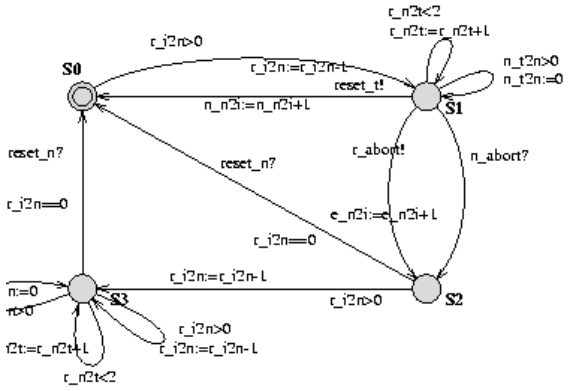

Figure 7: Timed automaton for the NormalActivity component.

The NormalActivity component is the part of the iC2C responsible for its functional aspects, that is, it processes service requests and answers them through notifications. Also, error detection mechanisms are implemented in the NormalActivity component. It is composed by two parts: the first one is responsible for the provision of service; it might be an off-the-shelf component or one created from the scratch. The other part is responsible for error detection, and the signaling of exceptions.

Functioning of the NormalActivity component: the NormalActivity component is initially idle, waiting for requests to arrive. When a request is received, it is processed. If necessary, the NormalActivity component may request services from components located in upper layers of the architecture. If a request is processed without exceptional notifications being sent or received by, the NormalActivity component requests the iC2C_top connector to end its activities and return to its initial state and sends a notification downwards the architecture with the results produced.

During the processing of a request, two kinds of problem might arise: internal and external. Internal problems are failures that manifest in the NormalActivity component, while it is attending a request. In this case, the NormalActivity component sends a request to the iC2C_top asking it to interrupt its activities and, at the same time, sends an exceptional notification to the iC2C_internal connector. External problems are failures that manifest outside the iC2C. These failures produce exceptional notifications which end up being received by the iC2C_top connector of the iC2C. The iC2C_top reacts by sending a notification to the NormalActivity component asking it to interrupt its activities and relaying the exceptional notification received to the iC2C_internal connector. The occurrence of any one of the two kinds of problem takes the iC2C to the abnormal activity mode.

Once in the abnormal activity mode, the NormalActivity component might receive service requests from the AbnormalActivity component, if these are necessary to recover the

system. These requests are processed in the same manner as they would be, were the system in the normal activity mode. They may even cause the NormalActivity component to send requests to other components located in upper layers of the architecture. Notifications produced are routed by the iC2C_internal connector to the AbnormalActivity component. When the recovery procedure ends, a request asking the NormalActivity component to return to its initial state is sent by the AbnormalActivity component and relayed by the iC2C_internal.

State	Description
S0	Idle. Waiting for a request.
S1	Normal processing of a request.
S2	Normal processing suspended after an exception.
S3	Recovering from an exception.

Transition		Specification			Description
From	To	Guard	Sync	Assign	
S0	S1	r_i2n>0		r_i2n:=r_i2n-1	A request for service is received.
S1	S1	r_n2t<2		r_n2t:=r_n2t+1	Sends a request to the top connector. Output queue size limited to 2.
S1	S1	n_t2n>0		n_t2n:=0	Process pending notifications, received from the top connector.
S1	S2		r_abort!	e_n2i:=e_n2i+1	Synchronizes with the top connector to suspend normal processing and sends an exception notification to the internal connector.
S1	S2		n_abort?		A notification to suspend normal processing is received from the top connector.
S2	S0	r_i2n==0	reset_n?		A request to synchronize at its initial state is accepted. There should be no pending request to process.
S2	S3	r_i2n>0		r_i2n:=r_i2n-1	A request for service is received.
S3	S3	r_i2n>0		r_i2n:=r_i2n-1	A request for service is received.
S3	S3	r_n2t<2		r_n2t:=r_n2t+1	Sends a request for service to the top connector. Output queue size limited to 2.
S3	S3	n_t2n>0		n_t2n:=0	Process pending notifications, received from the top connector.
S3	S0	r_i2n==0	reset_n?		A request to synchronize at its initial state is accepted. There should be no pending request to process.

3.5 AbnormalActivity Component

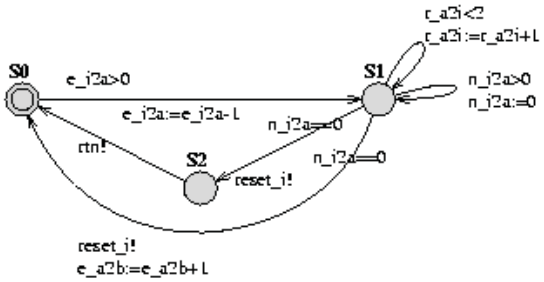

Figure 8: Timed automaton for the AbnormalActivity component.

The AbnormalActivity component is the part of the iC2C which encapsulates its error recovery mechanisms. While the system is functioning in the normal activity mode, the AbnormalActivity component remains deactivated. From the moment an exceptional notification is received on, the AbnormalActivity component is activated with the goal of making the system state consistent again.

Functioning of the AbnormalActivity component: initially, the AbnormalActivity component is idle. When an exceptional notification is received, the component is activated. It is important to note that the AbnormalActivity component is active only while the iC2C is operating in the abnormal activity mode. Once activated, the AbnormalActivity component tries to eliminate the error from the system state. To achieve this task, it might use its internal fault-tolerance mechanisms or request services from the NormalActivity component or from components in upper layers of the architecture. Response notifications are routed to it by the iC2C_internal connector.

In case it is not possible to recover the system state, the AbnormalActivity component throws an exceptional notification so that some component located in a lower layer of the architecture might try to correct the problem. At the end of the recovery process, independently of its success or failure, the AbnormalActivity component sends messages to the iC2C_internal connector requesting it and the other sub-components of the iC2C to return to their initial states.

State	Description
S0	Idle. Waiting for an exception notification.
S1	Handling an exception notification (error recovery).
S2	Error recovery was well succeeded. Resuming normal operation.

Transition		Specification			Description
From	To	Guard	Sync	Assign	
S0	S1	e_i2a>0		e_i2a:=e_i2a-1	An exception notification is received.
S1	S1	r_a2i<2		r_a2i:=r_a2i+1	Sends a request for service to the internal connector. Output queue size limited to 2.
S1	S1	n_i2a>0		n_i2a:=0	Process pending notifications, received from the internal connector.
S1	S0	n_i2a==0	reset_i!	e_a2b:=e_a2b+1	Sends a request to synchronize the internal connector at its initial state and sends an exception notification to the bottom connector. There should be no pending notifications to process.
S1	S2	n_i2a==0	reset_i!		Sends a request to synchronize the internal connector at its initial state, preparing to return to normal operation. There should be no pending notifications to process.
S2	S0		rtn!		Sends a request to the internal connector to return to normal operation.

3.6 Client and Server Processes

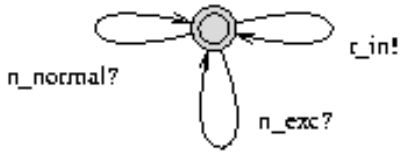

Figure 9: Timed automaton for a client process. This process uses some service(s) provided by an iC2C.

Transition		Specification			Description
From	To	Guard	Sync	Assign	
S0	S0		r_in!		A service request is sent to the iC2C.
S0	S0		n_normal?		A normal notification is received from the iC2C.
S0	S0		n_exc?		An exceptional notification is received from the iC2C.

Figure 10: Timed automaton for a server process. An iC2C uses some service(s) provided by this process.

Transition		Specification			Description
From	To	Guard	Sync	Assign	
S0	S0		r_out?		A service request from the iC2C is received.
S0	S0	n_t2i<2, n_t2n<2	n_in!		A notification is sent to the iC2C.

3.7 List of Messages

Message	Mode	Description
e_a2b	async	failure exception sent by the AbnormalActivity to the iC2C_bottom
e_i2a	async	failure exception sent by the iC2C_internal to the AbnormalActivity
e_n2i	async	failure exception sent by the NormalActivity to the iC2C_internal
e_t2i	async	external exception notification sent by the iC2C_top to the iC2C_internal
n_abort	sync	notification sent by the iC2C_top to the NormalActivity after an external exception is received
n_exc	sync	external failure exception sent by the iC2C_bottom connector to a client component
n_i2a	async	notification sent by the iC2C_internal to the AbnormalActivity
n_i2b	async	normal notification sent by the iC2C_internal to the iC2C_bottom
n_in	sync	notification sent by a server component to the iC2C_top connector
n_n2i	async	normal notification sent by the NormalActivity to the iC2C_internal
n_normal	sync	normal notification sent by the iC2C_bottom connector to a client component
n_t2i	async	notification sent by the iC2C_top to the iC2C_internal
n_t2n	async	notification sent by the iC2C_top to the NormalActivity
r_a2i	async	request sent by the AbnormalActivity to the iC2C_internal
r_abort	sync	control request sent by the NormalActivity to the iC2C_top after an internal exception occurs
r_b2i	async	request sent by the iC2C_bottom to the iC2C_internal
r_i2n	async	request sent by the iC2C_internal to the NormalActivity
r_i2t	async	request sent by the iC2C_internal to the iC2C_top
r_in	sync	service request sent by a client is accepted by the iC2C_bottom connector
r_n2t	async	request sent by the NormalActivity to the iC2C_top
r_out	sync	service request sent by the iC2C_top connector to a server component
reset_i	sync	control request to reset iC2C_internal to its initial state
reset_n	sync	control request to reset NormalActivity to its initial state
reset_t	sync	control request to reset iC2C_top to its initial state
rtn	sync	"return to normal" request sent by the AbnormalActivity to the iC2C_internal
start_t	sync	control request sent by the iC2C_internal to de block iC2C_top
sync_i	sync	control request to synchronize initial states of iC2C_bottom and iC2C_internal connectors

4 Implementation Issues

This section presents some issues relevant to the implementation of the idealised C2 component. These issues represent problems that might be found during the implementation of the iC2C and which are not directly related to its specification. For each of them, one or more possible solutions are outlined.

4.1 Notification-processing Policy

As mentioned in Section 3, notifications received by the iC2C_top connector are sent downwards the architecture only when a request is being processed by the iC2C. This policy is motivated by the fact that, when starting to process a request, an iC2C is always in its initial state. To attend a notification outside the context of a request, the receipt of new requests would have to be blocked until the processing of the notification ended. Two problems arise from this observation:

1. It becomes necessary to include some control between the connectors iC2C_bottom and iC2C_top to make the iC2C a mutual exclusion region for notifications and requests.

2. To know when the processing of a notification ends is hard because, unlike requests, the end of the processing of a notification does not necessarily trigger the sending of another notification.

It is easy to get round these problems simply by not allowing notifications to be attended outside the context of a request.

4.2 Notifications Enqueued in the iC2C_top Connector

Due to the asynchronous communication assumptions made by the C2 architectural style, a message sent by a component or connector might take an arbitrarily long time to reach its destination. Consequently, a request being processed by an iC2C may generate other requests for which the responses might arrive only after the original request has already been processed. Components may also send notifications due to the occurrence of asynchronous events, normal or exceptional.

As mentioned in Section 3.3, notifications are processed by the iC2C only when it is attending a request. Consequently, long time periods without requests being received make notifications wait for long time periods as well, even when they should be processed as soon as possible, as in the case of exceptional notifications. Since this behavior is undesirable, a mechanism that guarantees that notifications enqueued in the port of the top domain of the iC2C_top connector are processed even when no requests are received becomes necessary. A solution to this problem consists on stipulating a maximum amount of time without the arrival of requests which is considered acceptable. Whenever this time limit is reached, a notification is sent by the iC2C_top connector informing the iC2C_bottom connector about the situation. The iC2C_bottom then sends a dummy request so that some of the notifications enqueued in the port of the top domain of the iC2C_top may be processed.

Another interesting approach which also employs the concept of a maximum amount of time which is considered acceptable consists of making the iC2C_bottom connector check for timeouts, instead of the iC2C_top. Whenever the time limit is reached, the iC2C_bottom sends a dummy request upwards, through the iC2C. This approach has two advantages over the first:

- The code responsible for the solution of the problem is restricted to one connector, making the mechanism easier to implement and maintain. The iC2C_top remains unchanged, relaying notifications to the iC2C only when requests are being processed.
- Only one extra message is sent, which makes the overall process faster, specially if we take into account the fact that the communication inside the iC2C is asynchronous. Even in the case of synchronous communication, the first approach requires two messages, which produces a two times greater round-trip time in the worst case. This issue is very important if the sub-components of the iC2C are distributed throughout a network.

Both the approaches may be implemented internally or externally to the connectors. An external implementation has, as its main advantage, the fact that no additional complexity is put in the connectors. Code responsible for tasks such as timeout monitoring and the sending of dummy requests is deployed as separate components connected to the iC2C_bottom and, depending on the chosen approach, iC2C_top connectors. Moreover, the additional behavior defined in this section is not part of the internal protocol of the iC2C, presented in Section 3. This fact alone justifies a separate implementation, since it would only reflect the conceptual model.

4.3 Multiple Response Notifications

Depending on the application domain and the implementation of the application, the NormalActivity component of an iC2C may need to send, as a response to a request, multiple different notifications. In these cases, all the notifications should be packed in a single message, so that the other members of the iC2C which receive this message (the iC2C_internal and iC2C_bottom connectors) know that the processing of the request has ended (as specified in Section 3).

Since the client components of the iC2C hope to receive a response composed by multiple notifications, however, the single notification must be decomposed in multiple messages before it leaves the iC2C. Hence, a mechanism which achieves this task and sends the resulting notifications downwards the architecture must be incorporated in the iC2C.

The mechanism responsible for the decomposition of the response message into multiple notifications may be implemented internally or externally to the iC2C_bottom connector. As described in the previous section, an external implementation decreases the complexity of the iC2C_bottom and is also more faithful to the specification. The fact that a notification may be composed by multiple messages is related exclusively to the implementation of the iC2C.

4.4 Priority of Notifications in the iC2C_top

In our implementation of the idealised C2 component, normal and exceptional notifications are very similar; a message is passed from a component or connector to the other without regard for its type and both exceptional and normal notifications are implemented by the same class. In spite of this, they must be treated differently when it comes to processing. Inside the iC2C, exceptional notifications must always be processed before normal notifications. In this manner, errors are corrected quicker and the probability that an error affects a non-reliable component which was not, originally, involved with its occurrence is lower.

To guarantee that exceptional notifications are processed first, the port of the top domain of each element of the iC2C should have two queues: one for the normal notifications and another one for the exceptional ones. A notification in the *normal notifications queue* of a port is only processed if the corresponding *exceptional notifications queue* is empty.

4.5 Notifications Received Too Late

All notifications that enter the iC2C and all the requests that leave it pass by the iC2C_top connector. Due to the asynchronous communication assumptions made by the C2 architectural style, the response to a request might arrive at the iC2C after an arbitrarily long period of time, when it is not necessary anymore. Hence, if a request *Req* is being processed by the iC2C, it is important to guarantee that notifications generated by requests processed prior to *Req* do not interfere with its processing.

This guarantee is given by the iC2C_top, which acts as a filter and lets only notifications relevant to the request currently being processed enter the iC2C. During the processing of *Req*, the iC2C_top connector only delivers to the iC2C notifications corresponding to responses to requests sent during the processing of *Req*, or related to asynchronous events, such as exceptions. Other notifications are simply discarded.

4.6 Messages in the Internal Queues of the iC2C

When the iC2C finishes attending a request, it is necessary for it to return to its initial state. This way, the processing of a request *Req* is not affected by an error occurred while a request Req', prior to *Req*, was being processed. This leads to the conclusion that the state of an iC2C should not persist between successive requests. That is, if the state of an iC2C was S *before* the processing of request *Req*, it should also be S before another request Req' is processed. It should not matter whether Req' is attended before or after *Req*. It is easy to guarantee that this condition holds if we take into account exclusively the components and connectors which compose the iC2C. When the processing of a request finishes, however, notifications and requests may be waiting in the internal queues of the iC2C.

Before going into further detail, we will introduce some notation. Let N, A, T, I and B be shorthands for NormalActivity component, AbnormalActivity component, iC2C_top connector, iC2C_internal connector and iC2C_bottom connector, respectively. We represent the queue in a port of one of the domains of the component or connector Y, which receives messages sent by component or connector X, as $X2Y$, where $X, Y \in \{N, A, T, I, B\}$ and $X \neq Y$. For example, if component X is located in a higher layer in the architecture than

connector Y, then $X2Y$ would refer to the port in the top domain of connector Y, which receives notifications coming from component X.

Careful inspection of Section 3 reveals that few actions need to be taken in order to guarantee that, when the processing of a requests ends, all the internal queues of the iC2C are empty. During normal activity, we may assume that queues $B2A$, $A2B$, $A2I$, $I2A$, $I2T$ and $T2I$ are not used, simply by performing some filtering in the connectors iC2C_bottom, iC2C_internal and iC2C_top. This filtering is based exclusively on the type of the message and conforms with all the rules defined by the C2 architectural style. Queues $B2I$, $I2B$, $I2N$ and $N2I$ are guaranteed to be empty in the end of the processing by the construction of the protocol, since each request consists of exactly one message and, as mentioned in Section 4.3, the response to a request generates exactly one notification (which signals the end of the processing of the request).

The specification of the iC2C_top connector states that if the transition $S1 \rightarrow S1$, with guard $r\_n2t > 0$, is taken, all the requests waiting in queue $N2T$ will be sent upwards the architecture. Hence, to guarantee that queue $N2T$ is empty by the end of the processing of a request, it is necessary to explicitly state $r\_n2t == 0$ as the guard for the transition $S1 \rightarrow S0$ in the specification of connector iC2C_top. This is also applicable for transition $S1 \rightarrow S1$, with guard $n\_t2n > 0$, in the specification of the NormalActivity component, with respect to queue $T2N$.

During abnormal activity, the AbnormalActivity component sends a synchronous $reset\_i$ message to the iC2C_internal connector when the error recovery process is about to end. In the moment the $reset\_i$ message is received, the iC2C_internal connector sends the synchronous reset messages $reset\_t$ and $reset\_n$ to the iC2C_top connector and the NormalActivity component, respectively. These messages guarantee that queues $I2A$, $A2I$, $I2T$, $T2I$, $I2N$ and $N2I$ are empty when the iC2C returns to normal operation. Queue $B2I$ is not used during abnormal activity. According to Section 3.2 of the specification, queue $I2B$ is also not used, since any notifications received by the iC2C_internal connector are sent exclusively to the AbnormalActivity component (transitions $S4 \rightarrow S4$ with guards $n\_t2i > 0$ and $n\_n2i > 0$). Queues $N2T$ and $T2N$ are guaranteed to be empty by the time the error recovery process ends due to transitions $S3 \rightarrow S3$, with guard $n\_t2n > 0$, and $S5 \rightarrow S5$, with guard $r\_n2t > 0$, of the specifications of the NormalActivity component and the iC2C_top connector, respectively.

5 Conclusion

In this work we presented the specification of the idealised fault-tolerant C2 component. We adopted a formal approach for the specification, so that properties of the model built can be proved. The iC2C was modeled as a network of processes. Each process behaves according to a finite state machine where transitions are triggered by internal events and the receipt of messages.

The main goal of this document is to provide the guidelines needed by developers to implement the idealised C2 component. As such, besides the specification of the iC2C, we also discussed some issues relevant to the implementation which are not addressed by the

specification, such as the cleaning of the internal queues of the iC2C.

The properties which we aim at proving have not yet been proved, due to the great computational demands of the model checking tool we are using, Uppaal. In the near future, we intend on trying to prove the properties again by running Uppaal in a cluster of computers which provides a much greater computational power that a single machine.

Another goal for the near future is the actual implementation of the iC2C. This implementation aims at incorporating the concept of idealised fault-tolerant C2 component in the C2 framework[7, 13], a set of classes and interfaces which provide the basic infrastructure necessary for the construction of a C2-style application, so that developers already used to the C2 architectural style are able to make the systems they build dependable with little additional effort.

References

[1] T. Anderson and P. A. Lee. *Fault Tolerance: Principles and Practice*. Prentice-Hall, 2nd edition, 1990.

[2] Algirdas Avizienis. The n-version approach to fault tolerant software. *IEEE Transactions on Software Engineering*, 11(2):1491–1501, December 1985.

[3] J. E. Cook and J. A. Dage. Highly reliable upgrading of components. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'1999)*, pages 203–212, New York, NY, May 1999. ACM Press.

[4] Paulo Guerra, Cecília Rubira, and Rogério de Lemos. An idealized fault-tolerant architectural component. In R. de Lemos, C. Gacek, and A. Romanovsky, editors, *Proceedings of the Workshop on Architecting Dependable Systems, ICSE'2002*, Orlando, FL., May 2002.

[5] V. Issarny and J. P. Banatre. Architecture-based exception handling. In *Proceedings of the 34th Annual Hawaii International Conference on System Sciences*. IEEE, 2001.

[6] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1997.

[7] N. Medvidovic, P. Oreizy, and R. N. Taylor. Reuse of off-the-shelf components in c2-style architectures. In *Proceedings of the 1997 Symposium on Software Reusability*, 1997.

[8] N. Medvidovic, D. S. Roseblum, and R. N. Taylor. A language and environment for architecture-based software development and evolution. In *Proceedings of the 21st International Conference on Software Engineering (ICSE 99)*, pages 44–52, May 1999.

[9] Marija Rakic and Nenad Medvidovic. Increasing the confidence in off-the-shelf components: A software connector-based approach. In *Proceedings of the 2001 Symposium on Software Reusability*, pages 11–18. ACM/SIGSOFT, May 2001.

[10] B. Randell and J. Xu. The evolution of the recovery block concept. In *Software Fault Tolerance*, chapter 1, pages 1–21. John Wiley Sons Ltd., 1995.

[11] David Sprott. Componentizing the enterprise application packages. *Communications of the ACM*, 43(4):63–69, April 2000.

[12] R. N. Taylor, N. Medvidovic, K.M. Anderson, Jr. E. J. Whitehead, and J.E. Robbins. A component- and message- based architectural style for GUI software. In *Proceedings of the 17th International Conference on Software Engineering*, pages 295–304, April 1995.

[13] UCI. ArchStudio 3.0 homepage, 2002. http://www.isr.uci.edu/projects/archstudio.

[14] Gary Vecellio and William M. Thomas. Issues in the assurance of component-based software. In *Proceedings of the 2000 International Workshop on Component-Based Software*, Carnegie Mellon Software Engineering Institute, 2000.