Modelando un Sistema de Teleoperación de Robots en DisCo

Tesis de Licenciatura

Autor
Pablo E. Giambiagi
Universidad de Buenos Aires
Departmento de Computación

Co-directores
Reino Kurki-Suonio, TTKK, Finlandia
Rosita Wachenchauzer, UBA, Argentina

Julio, 1997

Abstract

En esta tesis construimos la especificación formal de un Sistema de Teleoperación de Robots conformado por un robot, un software de simulación y una cámara de video. Durante el desarrollo, prestamos especial atención a la derivación transformacional de las especificaciones y, en particular, a aquellos pasos de derivación que introducen aspectos relacionados con el diseño.
Nos interesa también la metodología, i.e., la organización de los pasos de transformación que se aplican en la derivación de la especificación del Sistema de Teleoperación de Robots, de modo que ese conocimiento pueda reutilizarse en casos similares.
El lenguaje elegido para la especificación es DisCo. Este lenguaje está basado en las ideas de "acciones conjuntas" (joint-action) de Back y Kurki-Suonio. Las pruebas de las transformaciones se realizan en el contexto formal de la Lógica Temporal de Acciones (TLA) de Lamport debido a que la semántica de DisCo está definida en este formalismo.
Comenzamos con una sencilla especificación del sistema que, a la vez, es lo suficientemente rica como para permitirnos expresar un importante requerimiento de seguridad. A partir de esta, se deriva una nueva especificación que toma en consideración las restricciones impuestas por la transmisión de información entre ubicaciones distantes entre sí, y la infactibilidad de eventos simultáneos en más de una de estas ubicaciones. El método utilizado, denominado "Combinación Sincronizada" ha sido utilizado en varias oportunidades en el pasado. Sin embargo, hacemos aquí un análisis de sus características y estudiamos cómo utilizarlo para probar propiedades de las especificaciones derivadas, a partir de las propiedades de la especificación inicial.

1 Introducción

Este trabajo es esencialmente un estudio práctico en el área de la especificación transformacional y formal de sistemas reactivos distribuidos, donde el objeto de estudio consiste de un sistema de teleoperación de robots basado en modelos. Esto último significa que el robot es teleoperado por medio de órdenes remotas emitidas desde un modelo del robot y su entorno (manipulados por un simulador especializado).

El Sistema de Teleoperación de Robots especificado aquí fue desarrollado e implementado como parte del Proyecto TELESAFE -integrado por la Universidad Tecnológica de Tampere (Finlandia), la Universidad de Louisville (EE.UU.), entes gubernamentales finlandeses, etc. Su principal objetivo fue el estudio de los mecanismos de seguridad necesarios para realizar teleoperación de robots.

Desde el punto de vista de la especificación formal del sistema, se prefirió un formalismo basado en acciones por sobre aquellos basados en procesos. Los formalismos basados en acciones se sostienen en la noción de acción, esencialmente una transición atómica en un sistema de transición de estados. La atomicidad impide la concurrencia de acciones y, de esa manera, se simplifica el estudio de las propiedades de los sistemas así definidos.

El marco lógico utilizado para analizar los sistemas de acciones está dado por la Lógica Temporal de Acciones (TLA) de Lamport [Lam94] y el lenguaje usado para especificarlos es DisCo (Distributed Cooperation, [BKs88], [Jar92], [JKs91], [JKSS90]).

Operacionalmente hablando, la noción de acción esconde toda comunicación entre procesos. Por ese motivo, las especificaciones deben sufrir una transformación capaz de hacer explicita dicha comunicación, antes de poder ser implementadas. En este trabajo, estudiamos diversos métodos de transformación y aplicamos aquel que, a nuestro juicio, resulta más conveniente en este caso.

1.1 TeleSys: Un Sistema de Teleoperación de Robots

TeleSys es un sistema -basado en modelos- para la operación a distancia de un robot. Los componentes del sistema son:

- Un robot industrial de seis (6) grados de libertad y una pinza neumática.

software de simulación) y un objeto coordinador (Telesys). Para un listado completo de esta especificación, consultar el apéndice A.

3.1 El Robot

Para modelar la combinación del robot Motoman y su controladora YASNAC, utilizamos una clase Robot que refinamos progresivamente para incluir varias de las restricciones impuestas por estos dos componentes de hardware.

La posición del robot (i.e., el estado de sus seis ejes y de la pinza) fue simplificada a un entero, ya que siempre podemos considerar que todas las coordenadas están así codificadas. El robot puede ser programado para realizar un movimiento lineal entre la posición actual y una posición dada, de manera que no tenemos que preocuparnos por conocer las posiciones intermedias y el control del robot para realizar dicho movimiento. Esto se refleja particularmente en la acción que representa el movimiento del robot (move_Robot) que cambia directamente la posición actual por la de destino, sin incluir puntos intermedios.

3.2 El Simulador (Telegrip)

En el proyecto TELESAFE, se decidió utilizar TELEGRIP^{TM} (Tele-Interactive Graphics Robot Instruction Program) como el software de simulación. TELEGRIP es un sistema de simulación y programación de robots "fuera de línea" que provee una interface de bajo nivel (LLTI) para interactuar con funciones definidas por el usuario. Nuestra modelización del simulador –a través de la clase Telegrip– sólo permite conocer el estado de la LLTI, i.e., si está activa o no, y en el primer caso, la posición del robot de acuerdo con el modelo en Telegrip.

3.3 La Cámara

En el proyecto TELESAFE se dedicó una cámara de video programable para detectar movimientos dentro del área de operación del robot, no producidos por él mismo, como una medida extra de seguridad. En la especificación definimos una clase Camera con tan sólo dos estados posibles. El estado *idle* representa la situación en que la cámara no está realizando ninguna operación y el estado *detecting* indica que la cámara está ejecutando su programa de detección de movimiento.

3.4 El Coordinador del Sistema

Una vez definidas las clases que representan al robot, el simulador y la cámara de video, las combinamos en un único sistema bajo la coordinación de otra clase encargada de mantener la consistencia entre aquellas (Notar asimismo que los elementos del entorno, incluyendo el usuario, no se modelan como objetos, sino como acciones).

```
class TeleSys is
    state *inoperative, operative(last_pos: integer);
    extend inoperative by
        state *camera_off, camera_on;
    end inoperative;
    buf: sequence integer;
    initially buf = <>;
end;
```

Como no hay forma de solicitar a la cámara su estado, este debe ser repetido en el coordinador. Además, es preciso utilizar una cola de posiciones (buf) para permitir que el modelo en Telegrip se mueva más rapido o más lento que el robot real.

3.5 Clasificación de las Acciones de Telesys

Agrupamos aquí las acciones del sistema en tres categorías que nos serán de utilidad en la derivación de una nueva especificación del sistema a partir de esta:

- Acciones de Arranque: Sitúan a Telesys en su estado operativo y vacían el buffer de posiciones.
- Acciones de operación: Modifican la posición del robot en el modelo, leen dicha posición, la transmiten y mueven el robot.
- Acciones de detención: Sitúan a Telesys en estado inoperativo ante pedidos del usuario o eventos de alarma.

3.6 La Propiedad de Seguridad

En teleoperación de robots es importante garantizar que, bajo determinadas situaciones de control, el robot nunca efectúe un movimiento a una posición no ordenada desde el modelo remoto. Hemos llamado a tal garantía "la propiedad de seguridad" que podemos expresar informalmente como:

> "El camino seguido por el robot es siempre un prefijo del camino seguido por el modelo en el simulador"

Para expresar esta propiedad en TLA, simplemente extendemos la definición del sistema Telesys con dos variables nuevas:

- *mpath*: El camino seguido por el modelo del robot (en Telegrip) tal como este ha sido recibido por Telesys (por medio de la acción *get_new_pos*) desde el último paso que satisfizo

$$Telesys.inoperative \wedge Telesys.operative'$$

- rpath: El camino seguido por el robot real desde el último paso que satisfizo la misma condición de arriba.

Con estas variables, la propiedad de seguridad se expresa en TLA como:

$$\Box\,(Telesys.operative \Rightarrow mpath = rpath \;\&\; Telesys.buf)$$

4 Nociones de Implementación

Analizando Telesys, la especificación del Sistema de Teleoperación de Robots desarrollada en la sección anterior, encontramos un número de indicadores que nos muestran que dicha especificación está "lejos" de una posible implementación utilizando los mecanismos requeridos por el proyecto, i.e., el uso de comunicación asíncrona entre los procesos locales y remotos:

- El sistema está distribuido en dos lugares posiblemente muy separados (espacialmente) entre sí: La ubicación local (Robot, controladora, y cámara) y la ubicación remota (Simulador y teleoperador).
- La especificación no hace distinción entre estas dos ubicaciones y define acciones (atómicas) que requieren la modificación simultánea de variables en ambas ubicaciones.
- Una implementación donde dichas acciones sean ejecutadas en forma atómica no es realista ya que el tiempo necesario para completar la comunicación de un dato de una ubicación a la otra podría causar la postergación de la respuesta a un evento de gran importancia desde el punto de vista de la seguridad en el área de trabajo.

Se impone entonces la idea de construir una nueva especificación del sistema que tenga en cuenta estas consideraciones y, a la vez, esté basada en la especificación inicial (enfoque transformacional) de modo de poder "extender" de algún modo sus propiedades a la nueva especificación. Es fundamental notar también que la especificación buscada deberá "partir" la atomicidad de aquellas acciones de TeleSys que modifican ambas ubicaciones simultáneamente.

4.1 Relaciones entre especificaciones

Dado que se intenta establecer una relación entre las dos especificaciones del sistema de modo de poder probar propiedades de la más concreta, por redución a propiedades de la más abstracta, estudiamos las relaciones descriptas en la literatura.

4.1.1 La Relación de Implementación

Dadas una especificación abstracta A y una especificación concreta C, se dice que C implementa a A si y sólo sí, toda traza de ejecución de C es una traza de ejecución de A [AL91]. En TLA, esta relación se expresa simplemente con la implicación lógica: $C \Rightarrow A$

Del estudio de esta relación, surge que:

- La relación de implementación fuerza a que, en toda traza, las variables visibles de C se modifiquen exactamente como lo harían en alguna traza de A.
- Además, cuando C parte la atomicidad de ciertas acciones de A, lo anterior ya no es posible. Es decir, la relación de implementación usual no podrá ser probada para nuestras dos especificaciones.

4.1.2 Agregación de Transacciones Distribuidas

Como una alternativa a la relación de implementación, consideramos el método de Agregación de Transacciones Distribuidas [PD96]. Este método consiste en la definición de una función de abstracción entre los estados de C y los de A, tal que a partir de toda traza de C se pueda construir, por la aplicación de la función a cada estado de la misma, una traza de A. Lo importante es que este método es de utilidad cuando existe un conjunto de "transacciones" que son atómicas en A, pero tienen una contrapartida no atómica en C.

El método plantea, además, los siguientes requisitos:

- Es necesario dividir las variables de C en variables de especificación y variables de implementación. Las variables de A deben ser exactamente aquellas variables de especificación de C.
- Para cada transacción de C debe haber un único punto de "commit", que es cuando la transacción modifica por primera vez una variable de especificación.

El método presenta una serie de desventajas para nuestro proyecto:

- La función de abstracción sólo sirve para establecer una relación entre C y A.
- Las propiedades que se pueden extender de A a C dependen fuertemente de la función elegida.
- El método requiere una definición precisa de la función de abstracción. Esto puede implicar mucho trabajo y sólo debería intentarse una vez que se conocen las propiedades a probar.
- El método requiere que las variables de A sean las variables de especificación de C. En nuestro caso, algunas variables de Telesys serán reemplazadas por un par de variables, una en cada ubicación.

- Al exigir que las transacciones tengan su punto de "commit" tan pronto como se modifica la primera variable de especificación, se impide que una transacción "interrumpa" la ejecución de otra.

La siguiente sección describe la relación que efectivamente nos servirá en la transformación de la especificación TeleSys.

4.2 Combinación Sincronizada

Se trata de una generalización del método de Agregación de Transacciones donde la función de abstracción es reemplazada por una relación binaria R.

Básicamente, se construye una nueva especificación $SYNC(C, A)$ donde cada estado del sistema contiene los valores de las variables de C y de A (adecuadamente diferenciadas). Además, las acciones de $SYNC(C, A)$ se obtienen de sincronizar cada acción de C con alguna acción de A o un paso de "stuttering".

La especificación $SYNC(C, A)$ define implícitamente la relación R, lo que contrasta con los requerimientos del método de **Agregación de Transacciones**.

La sincronización se puede realizar imponiendo restricciones y propiedades a la relación resultante R, guiadas por las propiedes de A que se quieren extender a C.

El método consta de tres fases interrelacionadas:

1. Combinación:

   Consiste en la construcción de la especificación $SYNC(C, A)$. Esencialmente, se trata de definir la sincronización

   $$g : Actions(C) \rightarrow P(Actions(A) \cup \{Unchanged_{Vars(A)}\}) \setminus \emptyset$$

   donde $Unchanged_{Vars(A)}$ es una propiedad satisfecha por cualquier paso de stuttering de A.

   El sistema sincronizado entonces se define como

   $$SYNC(C, A) \equiv Init_{SYNC} \wedge \Box[A_{SYNC}]_{Vars(SYNC(C,A))}$$

   donde

   $$\begin{array}{rcl} Init_{SYNC} & \equiv & Init_S \wedge Init_T \wedge R^* \\ Vars(SYNC(C, A)) & = & Vars(C) \cup Vars(A) \\ A_{SYNC} & = & SYNC_1 \vee \ldots \vee SYNC_n \\ n & = & |Actions(C)| \end{array}$$

   y para cada i, tal que $1 \leq i \leq n$,

   $$SYNC_i \equiv S_i \wedge (B_1 \vee \ldots \vee B_k) \wedge (R^*)'$$

   asumiendo que $g(S_i) = \{B_1, \ldots, B_k\}$.

   Esta especificación establece implícitamente una relación R entre los estados de C y los de A de la siguiente manera: Dado $(e_c, e_a) \in States(C) \times States(A)$, entonces $e_c R e_a$ sí y sólo si existe una traza de ejecución $\sigma =< \sigma_1, \sigma_2, \ldots >$ de $SYNC(C, A)$ y un índice i, tal que $\sigma_i = (e_c, e_a)$.

2. Verificación:

   Se verifica que cada traza de C pueda ser extendida a una traza de $SYNC(C, A)$. En otras palabras, hay que probar que $SYNC(C, A)$ satisface

   $$\Box(Enabled(S_i) \Rightarrow Enabled(SYNC_i)) \quad (1)$$

   para toda acción $S_i \in Actions(C)$.

3. Traducción de propiedades:

   La relación R es utilizada para relacionar una propiedad de C que se quiere probar, con las propiedades conocidas de A.

5 Particionado de TeleSys

Ahora estamos en condiciones de generar una nueva especificación del Sistema de Teleoperación de Robots a partir de TeleSys de modo que se tenga en cuenta la separación entre las ubicaciones remota y local. La nueva especificación, que llamaremos TeleSysLoc, se obtiene de la siguiente manera:

- Se distribuyen las variables de acuerdo a su ubicación. Algunas atributos de las instancias de Telesys tienen que ser duplicados en ambas locaciones.
- Aquellas acciones de Telesys que modificaban variables en ambas ubicaciones son reemplazadas por procesos (conjunto de acciones que cooperan para lograr el mismo efecto que la acción original). Esto sirve para simular el efecto de la comunicación asíncrona.
- Se construye un proceso por cada conjunto de acciones (de arranque, de operación y de detención) de Telesys.
- Se agregan nuevos estados para representar los estados intermedios de cada proceso.

El Apéndice A incluye el listado completo de la especificación TeleSysLoc.

6 Sincronización de TeleSys y TeleSysLoc

Entre las dos especificaciones del Sistema de Teleoperación de Robots obtenidas se construye una relación de Combinación Sincronizada siguiendo el método descripto en 4.2. Previamente se tradujo cada especificación a TLA y en ese mismo formalismo se escriben las acciones del sistema sincronizado *SYNC(TeleSysLoc, Telesys)* (ver Apéndice B). La sincronización es realizada teniendo como objetivo la prueba de la propiedad de seguridad para la especificación concreta TeleSysLoc.

Luego, la sincronización es verificada probando el invariante (1) para cada una de las acciones de TeleSysLoc que se sincroniza con alguna acción de TeleSys.

6.1 La Propiedad de Seguridad para TeleSysLoc

Una vez construida y verificada la especificación *SYNC(TeleSysLoc, Telesys)*, sabemos que:

1. $\Box(R.position = sR.position)$
2. $\Box(RTS.oper \Rightarrow TT.active.position = sTT.active.position)$

donde R, RTS y TT son variables de *TeleSysLoc*, mientras sR y sTT son de *Telesys*.

Dichas propiedades nos permiten probar sin mucha dificultad que: Si *impath* e *irpath* son las variables de TeleSysLoc cuyos valores representan respectivamente el camino seguido por el modelo y el camino seguido por el robot, entonces

$$\Box(mpath = impath \wedge rpath = irpath)$$

A partir de aquí es inmediato que podemos basarnos en la propiedad de seguridad de Telesys

$$\Box(Telesys.operative \Rightarrow mpath = rpath \& Telesys.buf)$$

para probar la propiedad de seguridad para TeleSysLoc:

$$\Box(L\_Telesys.oper \Rightarrow impath = irpath \ \& \ queue(L\_Telesys, R\_Telesys))$$

donde $queue(L\_Telesys, R\_Telesys)$ es la lista de posiciones en "camino" hacia el robot.

7 Conclusiones y Trabajo Futuro

7.1 Resultados

Dos especificaciones, con distinto grado de abstracción, fueron construidas para el Sistema de Teleoperación de Robots del proyecto TELESAFE. La especificación de más alto nivel, a pesar de su sencillez, nos permitió expresar una importante propiedad de seguridad en teleoperación (sección 3.6).

La especificación de más bajo nivel (TeleSysLoc) se obtuvo a partir de la anterior, particionando la atomicidad de alguna de sus acciones dentro del marco del método de Combinación Sincronizada (sección 4.2). El objetivo de esta transformación fue el de introducir en la especificación mecanismos relacionados con la transmisión de datos y control entre ubicaciones distantes entre sí.

Durante la prueba de la Combinación Sincronizada, detectamos varios errores en la especificación original de TeleSysLoc que de otro modo hubiera sido bastante difícil de detectar.

7.2 Implementación

El Sistema de Teleoperación de Robots, Telesys, fue implementado en el lenguaje de programación C, junto con bibliotecas de sockets TCP. La aplicación producida fue utilizada para estudiar el tema de seguridad en ambientes de teleoperación de robots.

Esta tesis también sirve como documentacion para dicha implementación.

7.3 Lenguaje de Especificación

Hemos aplicado el lenguaje Disco a un problema del "mundo real", es decir que no fue concebido como una prueba de las virtudes del lenguaje.

El lenguaje restringe el tipo de refinamiento de acciones que se pueden realizar. Esto nos forzó, en algunas circunstancias, a escribir dos acciones cuando hubiera sido perfectamente válido escribir sólo una.

Por otro lado, cuando quisimos derivar TeleSysLoc a partir de Telesys, nos vimos obligados a definir un sistema completamente nuevo. La versión 2.0 del lenguaje [Dis94] no provee ningún mechanismo para reutilizar TeleSys. Sin embargo, la versión 3.0 [Dis96] sí lo tiene, pero no fue utilizada porque, al momento de escribir esta tesis, no contábamos con una herramienta de simulación de especificaciones para esta nueva versión.

7.4 Particionado de la atomicidad de las acciones

En nuestro trabajo hemos sostenido la importancia de poder realizar refinamientos en la atomicidad de las acciones si es que estamos interesados en un enfoque transformacional para la especificación de sistemas distribuidos. Asimismo, al intentar aplicar estas ideas a nuestras especificaciones, analizamos diversos métodos incluyendo el reciente método de "Agregación de Transacciones Distribuidas" [PD96]. Hemos visto que este método no se ajustaba a nuestros propósitos. En su lugar, aplicamos éxitosamente el también conocido método de "Combinación Sincronizada" (4.2). Además, hemos prestado especial atención a la descripción y análisis del método, cosa que consideramos la principal contribución de esta tesis.

7.5 Trabajo Futuro

Algunas posibles continuaciones para este trabajo son:

- Verificación semi-automática de la Combinación Sincronizada
- Refinamiento de la especificación del Sistema de Teleoperación de Robots.
  - Inclusión de otras consideraciones de diseño como, por ejemplo, canales de mensajes asíncronos.
  - Serialización
  - Modelización de errores en las comunicaciones. Tolerancia a fallas.
- Aprovechamiento de las nuevas facilidades de DisCo 3.0

Modeling a Robot Teleoperation System in DisCo

Licentiate Thesis

Pablo E. Giambiagi
University of Buenos Aires
Department of Computer Science

July, 1997

Abstract

In this thesis we produce a formal specification of a robot teleoperation system consisting of a robot, a simulator application and a camera. In doing so, we concentrate on the transformational derivation of specifications and pay particular attention to those transformations which introduce design-related issues into the specifications.

We are also interested in the methodology, i.e., in organizing the transformation steps to be applied in the derivation of the robot teleoperation system specification so that they could eventually be of use in similar cases.

The specification is done using the DisCo specification language, which is based on the joint action approach of Back and Kurki-Suonio. The proofs of the transformations use Lamport's Temporal Logic of Actions, in which the semantics of the DisCo language is also defined.

We start with a simple but rich enough specification of the system and derive from it an specification that takes into account the fundamental restrictions imposed by the transmission of data among distant locations and the inexistence of simultaneous events. The method applied, called Synchronized Combination, has been used on several occasions before, though we give here a thorough analysis of its characteristics and study how to use it to prove properties of the derived specification from properties of the original, supposedly simpler, specification.

Preface

This case study on the specification of the Telesys Robot Teleoperation System originated as part of the TELESAFE Project, a collaborative research project on safety in robot teleoperation with the intervention of the Finnish State Technical Research Centre (VTT), Tampere University of Technology (TUT, Finland), the University of Louisville (USA) and various others Finnish governmental organizations and companies.

I participated in the TELESAFE Project first as a IAESTE Trainee (July-September, 1995) and then as a research assistant (October 1995 – January 1996), always at the Institute of Hydraulics and Automation (IHA/TUT).

After that and till June 1996, this work was funded by the Software System Laboratory (SSL/TUT) where I worked at the DisCo Group as research assistant.

Finally, this thesis was mostly written while I was working as Teaching Assistant at the Department of Computing of the University of Buenos Aires, Argentina.

Acknowledgments

Prof. Reijo Tuokko (IHA/TUT) and Prof. Reino Kurki-Suonio (SSL/TUT) are both mainly responsible for all the good achievements of this work. Prof. Tuokko game me the opportunity of visiting Finland in first place, of getting involved in the TELESAFE Project and of staying there longer than what was originally expected. My gratitude for Prof. Kurki-Suonio, who by the way got me interested into TLA, DisCo and action systems, goes well beyond what can be written here.

The personnel from the Institute of Hydraulics and Automation (TUT), VTT Safety Engineering and the Software System Laboratory (TUT) have all provided me with wonderful working environments. Jari Sulkanen, Pasi and Klaus Pekonen (from VTT), Pertti Kellomäki, Timo Aaltonen and Peeter Pruuden (from SSL/TUT) were great fun to work with.

I also want to thank the people from the Department of Computer Science at the University of Buenos Aires – specially its director, Irene Loiseau, and my supervisor, Rosita Wachenchauzer– for supporting me in so many ways.

Finally, I wish to express here my gratitude to all those beloved friends who accompanied me during this period abroad and at home, specially José Luis, Eva, Thilo, Anne, Roger, Victor, Silvano, Dominik, Andrés, Beatriz and Annamari.

Pablo E. Giambiagi
Buenos Aires
July, 1997

Contents

Chapter 1

Introduction

This thesis work is fundamentally a case study into the transformational formal specification of distributed reactive systems where the object of study is a model-based robot teleoperation system.

In model-based teleoperation, there is a master station containing a model of the robot and its environment. The model, which is made so as to represent the slave station as accurately as possible, includes knowledge of the kinematics of the robot and all necessary information of the objects nearby. Model-based teleoperation presents many advantages over traditional teleoperation, particularly because it allows a higher degree of precision in the movement of the robot and because it requires far less transmission bandwidth.

The robot teleoperation system specified here has been actually developed and implemented as part of TELESAFE, a comprehensive research project at the Institute of Hydraulics and Automation (Tampere University of Technology, Finland) involving researchers from different disciplines and whose main objective was the study of safety requirements in teleoperation environments. In this context, it was assumed that formal methods would provide an appropriate framework to gain considerable confidence in the properties of the delivered software.

There are essentially two complementing approaches to the specification of concurrent and distributed systems (such as the robot teleoperation system presented here): *process-oriented* and *action-oriented.* In the *process-oriented* approach processes execute in parallel and communicate with each other by sending and receiving messages, or through shared memory. One of the main disadvantages of this approach is that it is usually difficult to reason about the properties of the whole system based on the behavior of each of the processes.

On the other hand, the action-oriented approach is built around what are called *event-based action system formalisms.* These formalisms are founded on the notion of **action**, mainly a transition in a state transition system. Each action is an atomic change of system state where atomicity means that no concurrency of actions is allowed in the model. Thus, a parallel execution of an action system yields the same results than a sequential and nondeterministic execution, therefore simplifying the reasoning necessary to understand its properties.

In this work, we follow the action-oriented approach. The logical framework used to discuss and analyze action systems is that of Temporal Logic of Actions (TLA, [Lam94]) and the language used to specify them is DisCo ([BKs88], [Jar92], [JKs91], [JKSS90]).

From an operational point of view, the action system approach makes interprocess communication implicit within an action. As a result, actions are not necessarily suitable for direct implementation in their general form. Of course, this lets us build specifications without being restricted by implementation-oriented mechanisms. However, it will then be part of the designer's duty to transform the higher-level specification into a similar, lower-level one which is better suited for a direct implementation. In other words, the designer is responsible for making those implementation-oriented mechanisms explicit.

This transformational approach is supported by a suite of transformation methods that assist the designer. Each method must satisfy at least the following requisites:

- The method should be constructive. That is, it should guide the designer through a series of steps that result in the deliverance of the transformed specification. Those steps should have a precise description in order to avoid ambiguities as much as possible.
- The method should establish a formal link between the properties of the higher-level specification and those of the lower-level one in such a way that it makes possible to derive properties of the latter (usually more complex) from properties of the former (usually simpler).

It will be argued that, when transforming a specification in order to make explicit the interprocess communication mechanisms, actions need to be split. Of the available methods (for example, refinement mappings, forward and backward simulation, hybrid methods, etc.) many do not account for those transformations involving the splitting of action's atomicity. Therefore those methods are of little help in these situations.

Therefore we intend to analyze the available methods and apply the one considered to be the more appropriate to our case study. We also plan to explicitly describe the characteristics of the method used, including the kind of relations between abstract and concrete specifications that it can manage and how the properties of the abstract specification are used to prove properties of the concrete one.

The following section enumerates the main objectives pursued not only in this work but also in enclosing TELESAFE Project.

1.1 Objectives

The objectives of this work are manifold. Some of them are related to the objectives of the TELESAFE project itself, mainly the study of security in a teleoperation environment; others have to do with the problem of specifying a reactive system using a formal method and finally, others relate to the precise methodology employed in the development of the specifications and to the characteristics of the system specified.

- Objectives related to TELESAFE:
  - To construct a specification of a Robot Teleoperation System.
  - To implement the specified Robot Teleoperation System.
  - To analyze safety conditions in a teleoperation environment using the implemented system as a test-bed.
  - To be able to verify properties of the specified system that have to do with the security issues considered in the TELESAFE project.
  - To document the software produced and formally describe the behavior of the hardware pieces used in the system
- Objectives related to the use of formal methods:
  - To gain experience into the advantages and disadvantages of the use of formal methods in Software Engineering.
  - To apply the DisCo Method to the transformational derivation of the specifications.
  - To apply DisCo to a "real system" that has not been designed with the mere intention of experimenting with the language and its properties.
- Objectives related to the specification of this particular Robot Teleoperation System:
  - To survey the different implementation methods available in the literature and to choose an appropriate one to be applied to the Robot Teleoperation System

- To extend the transformation methodology wherever necessary, specially when considering the splitting of actions due to the differentiation of locations and the necessary non-simultaneity of the events occurring at different locations.

1.2 TeleSys: A Robot Teleoperation System

As has been said before, this thesis consists in the specification of a Robot Teleoperation System. This system, which we have called *Telesys* for short, consists of a Motoman robot, a Telegrip simulation environment, a programmable camera and the software that manages them all in cooperation.

We hereby give a brief description of all three hardware components of the system. Nevertheless, notice that a more precise definition of the characteristics of each of the hardware and software components will be given as part of the specification of *Telesys* in the following chapters.

1.2.1 The Telegrip Simulation Software

TELEGRIP^{TM}, Tele-Interactive Graphics Robot Instruction Program is a 3D robot simulation and off-line programming software developed by Deneb Robotics, Inc., USA. TELEGRIP is used mainly for planning, building, programming and controlling intelligent robots and telerobots. as well as their system applications, by means of integrated sensors, graphic representation and real-time reasoning. With this software, the user can produce a highly detailed model of the robot and its environment. It also provides an optional interface called LLTI (Low Level Teleoperation Interface) that allows the bi-directional transmission of the model state information in real-time.

1.2.2 The Motoman Robot and its YASNAC-ERC Controller

For this project we used a Motoman K10S industrial robot with six degrees of freedom situated in VTT Manufacturing Technology, Tampere, Finland (VTT is the Finnish Technical Research Center).

To control it, we used a standard YASNAC-ERC robot controller which supports a RS232 interface to the outside world. This controller gives the possibility to read joint values of the robot instead of reading only TCP-point values, which saves us from determining the inverse kinematics of the robot.

In its "Remote" mode, the Motoman controller has also some additional and useful functions. For example, servo power can be turned on and off and alarms can be read and cancelled. Regarding safety, this controller allows the installation of several stop mechanisms for emergency situations.

1.2.3 The Smart Camera MAPP2200

This video camera has a programmable unit able to perform different tasks like edge- and motion-detection. In the Telesys system, the Smart Camera is used (as a security mechanism) to detect, once teleoperation has been started, any movement inside the robot workcell not produced by the robot itself.

1.3 Overview of the Thesis Work

The thesis has been organized in the following way:

Chapter 2 introduces action systems, the theoretical framework used to reason about them (i.e., the Temporal Logic of Actions), and the language used here to describe them (i.e., the DisCo language).

Chapter 3 then gives an initial, simple but complete specification of Telesys, where there is no distinction between remote and local locations.

It is then argued, in Chapter 4, that the specification produced in the previous chapter is inadequate for a proper implementation, as it does not make explicit the communication between the locations. This suggests that a transformation step should be applied to the specification of Chapter 3. After describing several different methods and showing them to be insufficient for this problem, the **Synchronized Combination** method is presented. Our main contribution here is presumed to be, not the method itself (as it has been applied on many occasions before), but in its presentation.

In Chapter 5, a new specification for Telesys is given, which now takes into account the restrictions imposed by differentiating the local and the remote locations.

By using the Synchronized Combination method, the specification of Chapter 5 is related to that of Chapter 3 and an important *security property* is proved for the former system by *reducing* it to properties of the simpler, latter one. This is done in Chapter 6.

Finally, Chapter 7 lists the conclusions of the thesis work and suggests lines for further development.

Chapter 2

Formal Specifications

2.1 Actions Systems and TLA

In this chapter, action systems and a mathematical foundation for them are introduced[1].

In the traditional process-oriented view, a software system involving parallelism is modeled as a set of communicating sequential processes. This often leads to complex systems, whose properties are hard to understand and reason about.

The joint action approach developed by Back and Kurki-Suonio [BKs88] concentrates on actions instead of processes. Each action is an atomic change of system state, expressed as a set of assignments. Action systems provide a convenient way of specifying reactive and parallel systems. One can often use local reasoning, i.e., reasoning that involves only those parts of the system that are affected by an individual action.

2.1.1 Action Systems

An action system consists of a set of state variables, and a set of rules, called actions, that tell how the values of variables may change. There are basically two ways of looking at an action system: an operational view and a logical view. We first take an operational view, and later introduce a logic (TLA) that can be used for the mathematical treatment of action systems.

2.1.2 Operational View of Action Systems

An action system can be regarded as an abstract machine consisting of a set of state variables, and a set of actions, where each action consists of a guard and a body. The guard is a boolean-valued expression involving state variables, and the body is a set of assignments to state variables.

The system starts in some initial state. As time passes, actions are executed, changing the system state accordingly. Actions are selected for execution nondeterministically, the only restriction being that the guard of an action must be true in order to be executed.

There is no notion of real time, only the ordering of actions is considered. The execution of an action is atomic, meaning that once the execution of an action has been started, it cannot be interrupted or interfered by other actions. The execution model is interleaving, i.e., only one action at a time is being executed. Note that this does not necessarily need to apply to the "reality" that is being modeled. If two actions do not refer to the same state variables, they may be executed in any order, or even simultaneously. The interleaving model is adopted, because it allows for simpler reasoning.

[1]The contents of this chapter have been extracted in their majority from Pertti Kellomäki's Licentiate Thesis [Kel94].

The following is a concrete, albeit not very useful, example of an action system which allows x to be incremented by one at any time, and to be divided by two when it is even:

$$\begin{array}{c} x \in N \\ \textit{when } \textit{true } \textit{do } x \leftarrow x + 1 \\ \textit{when } \textit{even}(x) \textit{ do } x \leftarrow x/2 \end{array} \tag{2.1}$$

When we use an action system as a specification of a system, we do not imply that the actual implementation should be an action system. We are only interested in the sequences of states resulting from executing the system. Any mechanism that produces (in some sense) the same sequences of states suffices as an implementation.

2.1.3 Temporal Logic of Actions

Action systems can be conveniently described and analyzed with Lamport's Temporal Logic of Actions (TLA) [Lam94]. We briefly introduce TLA, omitting some details that are not needed for the purposes of this case study.

State, State Functions and State Predicates

We assume that there exists an infinite number of variables, such that there is a unique value for each variable in each state. A *state* is a function mapping variable names to values.

A *state function* is an expression built from variables and values. A *state predicate* is a boolean-valued state function. The value of a state function in a given state is the value obtained by substituting variable names with their values in the given state, and evaluating the resulting expression under the ordinary mathematical interpretation.

Actions

An *action* is a boolean-valued expression built from values and unprimed and primed variables. An action represents a relation between states, where unprimed variables represent the values of variables in one state, and primed variables represent the values in another state. A pair of states is called a *step*, and a pair of states satisfying action A is called an A-step.

A TLA action embodies both the guard and the body of an action in the operational interpretation. For example, TLA equivalents of the actions in the system given in 2.1 are

$$x' = x + 1 \tag{2.2}$$

$$\textit{even}(x) \wedge x' = x/2 \tag{2.3}$$

A boolean-valued expression P involving only unprimed variables can also be interpreted as an action. A pair of states satisfies P iff the first state satisfies P.

The state predicate $\textit{Enabled}(A)$, defined for any action A, is true for a state s iff there exists a state t such that (s, t) is an A-step. The *Enabled* predicate corresponds to the guard of the action in the operational interpretation.

Temporal Logic

The temporal properties of an action system are dealt with by considering the sequences of states that can be taken under the operational interpretation of the system. An infinite sequence of states is called a *behavior*, and TLA provides means of making statements about such sequences. Finite sequences can be extended to infinite sequences by repeating the final state indefinitely.

Behaviors are associated with temporal ordering. The first state of a behavior is the initial state of the system, and the rest of the sequence represents the state of the system at different times.

Temporal Operators Temporal formulas are formed by using the unary operator $\Box$ (always), and the boolean operators. A temporal formula is interpreted as an assertion about behaviors. Let F be a boolean-valued formula not involving temporal operators, i.e., a state predicate. Its meaning $[\![F]\!]$ is that F holds for the first state of a behavior. We denote the boolean value that F assigns to a behavior σ with $\sigma[\![F]\!]$.

The formula $\Box F$ asserts that F holds for all states of a behavior, i.e., that F is always true. Let $< s_0, s_1, \ldots >$ denote a behavior whose first state is s_0, second state is s_1, etc. We can now define $[\![\Box F]\!]$ in terms of $[\![F]\!]$ as follows:

$$< s_0, s_1, \ldots > [\![\Box F]\!] \doteq \forall n \in N :< s_n, s_{n+1}, \ldots > [\![F]\!] \tag{2.4}$$

For convenience, we define the additional temporal operators $\Diamond$ (eventually) and $\rightsquigarrow$ (leads to):

$$\Diamond F \doteq \neg\Box\neg F \tag{2.5}$$

$$F \rightsquigarrow G \doteq \Box(F \Rightarrow \Diamond G) \tag{2.6}$$

The formula $\Diamond F$ states that F is true for some state of a behavior. The formula $F \rightsquigarrow G$ states that, whenever F is true, G is true then or at some later time.

By allowing actions in temporal formulas, we can describe action systems. We interpret action A as an assertion about the first step of a behavior, and $\Box A$ as an assertion about all steps in a behavior. Thus, a behavior satisfies A, iff the first step in the behavior is an A step, and it satisfies $\Box A$, iff every step is an A-step.

Adding stuttering to actions A step (s, t) in which all variables in a given set U have the same values in both s and t is called a U-stuttering step.

It is sometimes convenient to be able to express a step that is either a A-step or a U-stuttering step. So we define

$$[A]_U \doteq A \vee (U' = U)$$

where $U' = U$ is shorthand for $\forall x \in U : x' = x$.

Another useful notation allows us to express those executions of action A where stuttering is ruled out:

$$< A >_U \doteq A \wedge (U' \neq U)$$

2.1.4 Describing Action Systems with TLA

It is now easy to describe action systems with TLA. We give the initial condition with a state predicate $INIT$, and the actions $A_1, A_2, \ldots, A_n$, corresponding to the actions of the action system. Ignoring all fairness assumptions, the action system can be understood to define a TLA formula of the form

$$S \equiv INIT \wedge \Box[A]_U \tag{2.7}$$

where A is the disjunction of actions A_i and U is the set of all variables manipulated in the system. We thus identify an action system with its encryption in TLA, and use "action system S" and "TLA description of action system S" interchangeably. In fact, the Temporal Logic of Actions is defined to be the temporal logic whose elementary formulas are predicates and formulas of the form $\Box[A]_U$.

Safety and Liveness Properties

A specification of the form 2.7 only specifies what is allowed to happen. As long as no illegal steps are taken, the specification is satisfied. No requirements for executing any specific action is expressed by the specification, so a behavior where no state changes take place satisfies it trivially. The properties expressed by this kind of specifications are called *safety properties.* Informally, a safety property states that nothing "bad" will ever happen.

If we want a specification to express something about the progress of the computation, its *liveness properties*, we need to state the liveness conditions explicitly. Liveness properties are usually of the form "eventually X", i.e., something "good" will eventually happen. See [AS85] or [Ks96] for a precise definition of these properties in terms of behavior prefixes.

Any temporal property of a specification can be stated as a conjunction of safety properties and liveness properties [AS85].

In TLA, liveness conditions of concurrent algorithms are expressed by *fairness properties.* The execution is fair to an action if the action is guaranteed to be executed under certain conditions. *Weak fairness* with respect to an action asserts that the action must be executed if it remains possible to do so for a long enough time. That is, the action must eventually either be executed, or become disabled. *Strong fairness* with respect to an action asserts that the action must be executed if it is possible often enough to do so ([Ks96] shows how to express these fairness properties in TLA).

Canonical Form

A TLA formula describing a system can always be written in the canonical form

$$INIT \wedge \Box[A]_U \wedge F$$

where $INIT$ gives the initial state, A is the disjunction of all actions and F is a conjunction of fairness conditions.

2.1.5 Open vs. Closed Systems

When reasoning about a reactive system, one often needs information about the behavior of the environment of the system. Since the environment can also be viewed as a reactive system, it is natural to model the environment in the same formalism as the system itself. This leads to a *closed system approach*, where the specification describes both the environment and the system. In contrast, the *open system approach* specifies only the system. Naturally, to make any statements about the behavior of the system, one has to make assumptions about the environment. However, these are not considered as part of the specification and they may be expressed in a different formalism.

TLA can be used to describe and reason about both open and closed systems.

In this thesis, we have followed the *closed system approach.*

2.2 The DisCo Language

The DisCo specification language ([JKSS90], [JKs91]) is based on action systems, with additional features to aid in modularization and incremental specification. The semantics of the language is based on TLA. We now present enough of the language for the purposes of the specification of *Telesys*, the Robot Teleoperation System. A more thorough discussion of the language can be found in [Jar92].

A specification environment has been built for the DisCo language. Specifications written in DisCo can be executed and animated using a graphical tool [Dis94][Dis96].

2.2.1 Related Approaches

Nondeterminism is an essential part of action systems. The specification of the system only tells which of the actions are enabled and one of them is nondeterministically chosen for execution.

Dijkstra uses *guarded commands* [Dij76], which are based on nondeterminism. When a set of guarded commands is executed, one of the commands whose guards evaluate to true is nondeterministically chosen for execution. However, Dijkstra uses guarded commands as building blocks of terminating sequential programs, and no assumptions about whether a given guarded command will be executed or not are given.

The execution model of Unity [ChM88] is very close to actions systems. A program consists of a set of statements and each of the statements is executed infinitely often. Every assignment is conditional, so some of the statement executions might not change the state of the system. Unity also provides superposition and composition of systems, although in a way that is slightly different from DisCo's.

2.2.2 Introduction to the Language

A DisCo specification consists of a set of systems, each system describing a part of the whole, or refining some other system. The basic methods of building new systems are composition of two or more separate systems and refinement of systems by superposition.

Composition of systems allows for bottom-up design, where parts of the complete system are first specified separately and later combined to form a new system, possibly synchronizing some of the actions. A typical example of this is the combination of a system specifying stations and a system specifying a bus, to produce a new system that specifies a network of stations.

Superposition is used for stepwise refinement, or top-down design of systems. By using superposition, one can add new state components, extend actions to handle the new state components, or add new actions. An important application of superposition is to strengthen the guards of previously introduced actions by adding new conjuncts, thereby constraining the allowed behavior. For example, we might first model a network of stations by letting any station send at any time. Later we could impose a ring discipline by allowing only the station that has the sending token to send.

The semantics of the specification language guarantees that safety properties are preserved in composition and superposition. Liveness properties, on the other hand, are not always preserved.

Classes and Objects

The universe of a DisCo specification consists of a set of *objects*, which are instances of *classes*. An object has a distinct identity and can participate in actions.

The attributes of an object can be simple values (such as integers or truth values), sets of simple values, sequences of simple values, or states. A state is an enumeration which can be used for implementing state machines.

As with other object-oriented methods, inheritance can be used for building more complex classes from simpler ones. Discussion of inheritance is omitted here, as we do not use it.

A simple example of a class definition is the following specification of an input device:

```
class Device is
    state waiting, data_ready;
    data: integer;
end;
```

Actions

Disco Actions correspond to actions in an action system roughly in the same way as subroutines correspond to ordinary blocks of code in a programming language. A DisCo action is a template,

which tells how the objects participating in the action change. For example, an action to assign a value to the data field of a device object could be written as

```
action read(d: integer) by D: Device is
when D.waiting do
   →D.data_ready;
   D.data := d;
end;
```

Action *read* illustrates a number of points about actions. An action consists of a *name*, and optional *parameter list*, a *list of participants*, a *guard* and a *body*.

When an action is executed, the values for parameters are chosen nondeterministically. A typical use for parameters is the one in action read, where we want to indicate that the data field is changed, but we do not yet want to fix the value that it will contain. In later refinements of the system, restrictions can be imposed on the values that *data* can contain.

The list of participants tells what kinds of objects may participate in the action. When an action is executed, the formal participants are replaced by the actual objects participating in the action. The body of the action then tells how the attributes of the participants are changed.

An action can only be executed if parameters and participants can be found so that the guard is satisfied. The guard can refer both to the parameters and to the participants of the action. It is also possible to refer to the global state by using universal or existential quantification over classes.

The execution of an action is atomic, which means that it cannot be interrupted by other actions once it has been started. Only one action at a time is (logically) being executed, so no mutual exclusion problems can arise while executing an action. As with action systems in general this does not rule out non-interfering actions happening simultaneously in the "reality" being modeled.

Systems

Classes and actions are collected into systems, which are used for modularizing specifications. Systems can be combined, or a system can be refined by adding new actions and classes, or modifying existing actions and classes.

When two or more independently defined systems are combined into a new system, it is possible to synchronize actions from different subsystems. When actions are combined the guard of the resulting action is the conjunction of the guards of the original actions, and the body is a catenation of the original bodies.

Stepwise refinement of a system can be done by importing an existing system into a new system, and refining the classes and actions of the imported system. The refinements are based on superposition, i.e., the new properties are defined on top of the old properties. The language takes care to avoid conflicting refinements.

It is also possible first to make independent refinements of a system and later combine these into one system again. We do not consider this case here, as such combinations are not used in this case study.

Actions Refinement and Class Extension

The syntax of refinement of actions is illustrated by the following refinement of action *read*:

```
refined new_read of read
by ...
when ... d > 0 is
   ...
end;
```

The participants, guard and body of the original actions are referred to as "..." in the refinement.

Classes can be extended, i.e., new components can be added. Extension of the class Device illustrates this:

```
extend Device by
    id: integer;
end;
```

Assertions and Initial Conditions

Actions, classes and systems can all include assertions and initial conditions. An initial condition expresses a condition that should hold initially, and an assertion expresses a condition that should hold invariantly.

Assertions and initial conditions serve two purposes: on one hand they can be checked when simulating the system, and on the other hand they inform the reader of a specification about intended initial conditions and invariants of the system.

2.2.3 DisCo and TLA

It is relatively straightforward to map DisCo to TLA. We do not attempt to give the complete semantics of DisCo in terms of TLA (see [Jar92]), but outline how the mapping works for the cases we are interested in here.

Objects

TLA does not include the notion of objects as such, it only has variables. Objects can be represented as arrays of variables, the index serving as the identity of the object. However, in this thesis, we only need one instance per class, which allows us to represent each DisCo object by a compound variable where each component is accessed by means of the usual dot notation.

Actions

A DisCo action corresponds to a TLA action where the participants of the DisCo action are existentially quantified. Again, this is a general solution. In this particular work, though, we can simplify this as we know that there is only one instance of each class.

For example, if we know there is only one possible instance of class A, the action

```
action inc by obj: A is
when true do
    obj.i := obj.i + 1;
end;
```

corresponds to the TLA action

$$obj.i' = obj.i + 1$$

Chapter 3

Specifying TeleSys

In this chapter we give an initial, simple but complete specification of Telesys, where there is no distinction between remote and local locations. It is also worth noticing that all the design decisions taken in developing the following specifications were guided by the underlying objective of producing an specification dealing fundamentally with the communication among the different components of the system, rather than with the internal details of any of them.

We first develop specifications for the Robot (an incremental specification in fact), the Telegrip simulation software and the Camera. And then, all three components are tied together by means of a new object, called Telesys too, which represents the controlling software.

Notice that we have also decided to specify Telesys as a closed system, i.e., a system which models the environment actions as well as the internal actions. As a result of this, for example, the robot object variables are actually the robot controller variables. The controller is thought of as being part of the system. The interface between the controller and the environment is treated in a very abstract way, by expecting environment actions to directly access controller variables. Similar decisions have been taken for modeling the effect of the environment over the other components.

3.1 The Robot Subsystem

We proceed to define a model for the Motoman Robot subsystem used in the **TELESAFE** project (called "the Robot" from now on). As we are building a model for a concrete robot, we must take into account the restrictions imposed by this particular piece of hardware/software that results from the combination of the Robot and its **YASNAC ERC** controller.

3.1.1 A simple initial system

Classes

From a very abstract point of view, a Robot consists of a position and a state of movement:

- Position:

  The Robot used for the **TELESAFE** experiments has 6 joints and a gripper. Each joint is servo-operated by its own step-motor whose position is uniquely determined by the "pulse" value (an integer in an appropriate range). The gripper is pneumatic and can be in any of two positions, either "open" or "closed". This means that the complete spatial situation of the robot can be described by an integer 6-tuple plus a boolean value.

  However, we will abstract away the difference between the various joints and the gripper. Instead, our model will use a single non-negative integer value to represent the Robot position in space because, in fact, we are not interested in the precise ("real") position of

the Robot, but only in having the possibility to uniquely identify each position.

$$position : integer;$$
$$assert\ position >= 0;$$

- State of movement:

  The robot controller is responsible not only for the correct positioning of the robot in space, but also for its movement from one position to the other. This frequently involves complex calculations that are outside our scope of interest. Furthermore, the **YASNAC ERC** controller allows the user to define different types of movements (linear, circular, etc.) but during the experiments, just linear movement was used.

  After analyzing the **YASNAC ERC** Controller capabilities, it was decided that, to begin with, two states had to be differentiated. The robot can be either still or moving towards a destination following a linear trajectory, and only in the former state the controller will accept any order to move the robot to another point in space, or change the state of the gripper.

  state $*still,\ moving(destination : integer)$;

  This results into the following class definition for a Robot object:

```
class Robot is
    state *still, moving( destination: integer );
    position: integer;
    initially position >= 0;
    assert position >= 0;
end;
```

Actions

Ideally, the robot controller will allow the user to perform the following operations:

- ask for the current robot position,
- set the robot in motion towards a position provided by the user, and
- stop the robot.

The **YASNAC ERC** controller used in the experiments acts always as a server, answering demands from the user (either the operator or the teleoperation software). This means that there is no means by which the controller can inform that the robot has just reached its destination and is ready to accept a new position to move to. On the contrary, it is the user's own responsibility to inquire the controller appropriately as to detect the correct moment where a new movement can be ordered.

Since we are interested in specifying the communication properties of the system, it is undesirable to introduce such a level of detail into this initial model (we mean the necessary loop to wait for the robot to reach its destination).

Then the only events we are interested in are:

- The robot is set in motion towards a new destination.

```
action set_Destination( p: integer)
by r: Robot is
when r.still and p >= 0
```

```
do
    → r.moving( p );
end;
```

- The robot reaches its current destination as a result of completing its programmed movement.

```
action move_Robot
by r: Robot is
when r.moving
do
    r.position := r.moving.destination;
    → r.still;
end;
```

- The robot is stopped.

```
action stop_Robot
by r: Robot is
when true
do
    → r.still;
end;
```

Notes
Refer to Appendix A (page 77) for the complete listing of system Robot.

3.1.2 Adding signals to the Robot controller

The are several reasons why the Robot might be stopped. We would like then to be able to distinguish, at least, a simple stop after successfully reaching destination, from any error situation and from an stop command given by the user (refer to action stop_Robot in the previous section).

In order to do this, we present a new system, which has been called Robot_Signal and is a direct refinement of the Robot system defined above.

```
system Robot_Signal
  import Robot;
```

Classes
The "still" state is extended to distinguish between the following situations:

1. The Robot stops after completing a correct movement (imported action move_Robot)
2. The Robot stops after receiving a signal.

```
extend Robot by
    extend still by
```

```
        state *idle, signaled;
    end still;
end;
```

Actions

First, we refine action *move_Robot* in order to correctly reflect the event of the robot stopping after successfully performing a movement:

```
refined move_Robot is
when ...
do
    ...
    → r.still.idle;
end;
```

Every time a signal is received by the robot controller, the robot is immediately stopped. Also if the robot controller has received a signal, then the signal has to be cleared before the robot can perform any movement.

Two new events are considered:

1. Turning a signal on (set_Signal).

```
action set_signal
by r: Robot is
when true
do
    → r.still.signaled;
end;
```

   Note that signals are not distinguished. However, there can be several different types of signals, thus action set_Signal is always enabled.

2. Clearing all signals (clear_Signal).

```
action clear_signal
by r: Robot is
when r.still.signaled
do
    → r.still.idle;
end;
```

Notes

It could be argued that we should have refined action *set_Destination* as well, as it is expected that the Robot won't be ready to be moved before a signal has been adequately cleared. But, as the process needed to clear each signal depends on the type of the signal, we will now postpone this refinement (see page 20).

Refer to Appendix A (page 78) for the complete listing of system Robot_Signal.

3.1.3 Applying the restrictions imposed by the Robot controller

At this point in the development of a specification for the Robot system, we would like to introduce some restrictions derived from the actual controller used in the **TELESAFE** project. First, we study the particular signals that the controller can recognize and, afterwards, we analyze the different controller commands and their availability during the different controller states. This information will later be used to define a refinement of system *Robot_signal*, which we have called *Robot_DifSig*.

Let's separate now the different possible signals according to the documentation provided by the manufacturer of the **YASNAC ERC** controller[1]. This documentation states that the controller reacts to four different signals:

1. External Hold

   An *External Hold* signal is raised whenever the robot is *operating* (i.e., it is moving) and occurs any of the following:

   (a) The Stop Button at the controller panel is pressed,

   (b) The Stop Button at the T-Box (a hand-held command console) is pressed,

   (c) An appropriate signal is received through the "External Hold Line". This line was used in the **TELESAFE** experiments to provide the operators of the robot with a wrist-radio-transmitter that would allow them to stop the robot when they are not close to any of the consoles.

2. Emergency Stop

   Notice that the External Hold signal can only be activated while the robot is moving. On the other side, the Emergency Stop signal can be activated at any moment and it is reserved for particularly critical situations. During the **TELESAFE** experiments, the Emergency Stop signal was used in conjunction with a modem connected to the telephone network. It was argued that, due to the latency of the Internet connection, a complementary remote control mode was needed.

   Immediately after an Emergency Stop is recognized by the controller, the servos operating the robot are turned off (something which does not occur after any other signal).

3. Command Hold

   This signal is raised when the robot is stopped by a software command.

4. Alarm

   This signal can result from an error internal to the controller or, for example, the impossibility to reach a certain position where the robot has been instructed to move to.

We now proceed to analyze how each signal affects the enabledness of the actions in *Robot_signal*, as we know that for each of the signals above, different restrictions apply to its activation and deactivation, and the commands that can be executed while it is active.

Table 3.1 was extracted from the **YASNAC ERC** documentation and gives an example of these restrictions. The table only shows the rows corresponding to commands actually needed for the implementation of the Robot Teleoperation System. The following is a brief explanation of the commands accepted by the **YASNAC ERC** and their significance for the design of Telesys:

[1] It is important to notice that the information provided by the manufacturers of the Motoman Robot and the YASNAC ERC were by no means complete. Therefore, the results of these studies and their validation in practice, as well as their description in DisCo, were doubly appreciated by the owners of the hardware (i.e. VTT) and the institutions involved in the TELESAFE project (i.e. TTKK/IHA).

Command Name	Robot Stopped	Robot Running	Panel Holding or After Stop	Command Holding	Alarm/ Error
START	O	-	-	-	-
HOLD	O	O	-	O	O
RESET	O	O	-	O	O
DELETE	O	-	-	-	-
SVON	O	O	-	O	-
RALARM	O	O	O	O	O
RPOSJ	O	O	O	O	O
RSTATS	O	O	O	O	O

O : Possible - : Not possible

Table 3.1: YASNAC ERC Status and Command Availability

- START: instructs the controller to run a program, which is the only way to move the robot. This means that the START command is the one used to implement action *set_Destination.*

- HOLD: Turns on/off the Command Hold signal. In the first case, it stops the current program, as well as the robot movement. In the other one, it clears the Command Hold signal. Used in the implementation of actions *hold_on* and *set_Destination.*

- RESET: Clears an Alarm signal (check actions *clear_alarm1* and *clear_alarm2* in the following sections).

- DELETE: Deletes a robot program from the controller's memory. This is needed so as to be able to upload the next robot program. Used in the implementation of action *set_Destination.*

- SVON: Turns servo power on/off. Used in the implementation of actions *set_Destination*, *clear_emergency_stop1* and *clear_emergency_stop2* (check the following sections for their definitions).

  The following commands will be used in the implementation of several actions belonging to the Telesys system (see page 22).

- RALARM: Reads errors and alarm codes.

- RPOSJ: Reads the current position data in joint coordinates.

  Notice that the state of the gripper cannot be read from the controller data. Therefore, this information must be maintained by Telesys itself.

  The different treatments of joint coordinates and gripper status has been abstracted away in our model but, of course, it had to be taken into account during the implementation phase.

- RSTATS: Reads running status, alarm/error status and servo status.

Classes

As signals of different type can be raised at the same time, the state of the robot has to be extended with the necessary information to inform which signals have been raised and which have not. To do so, we first tried extending the still.signaled state with four new sub-states, one for each signal, but this approach resulted in a quite clumsy code, as we had to define two new identifiers for each signal: one identifier stated a raised signal condition while the other indicated a cleared condition. To avoid this unnecessary complexity, we chose to represent them

as boolean variables, i.e., one variable for each signal[2].

```
extend Robot by
    emergency_stop: boolean;
    external_hold: boolean;
    command_hold: boolean;
    alarm: boolean;
    initially not(emergency_stop or external_hold or
                  command_hold or alarm);
end;
```

Notice as well that these four variables are of interest only while the robot is in the *still.signaled* state. However, we have also decided not to define them as extensions to this state, because the notation *r.command_hold* was easier to manipulate than *r.still.signaled.command_hold*. The price of this simplification in notation is that we have to verify the following:

```
assert not(emergency_stop or external_hold or command_hold or alarm) or still.signaled;
```

which means that whenever any of the signal variables is true, then the robot is in the *still.signaled* state.

Actions

For each signal variable, we provide an action to set it on. Remember that the External Hold and Alarm signals could only be activated while the robot is moving. Therefore, as an example, we have:

```
refined set_alarm of set_signal is
when ... r.moving
do
    ...
    r.alarm := true;
end;
```

Also for each signal variable, we have to create an action to clear it. However, whenever a signal is cleared, we have to change the robot state accordingly. This cannot be done directly in **DisCo**, as we need to change a state defined in an imported system (Robot_Signal).

For example, we would have liked to define the action to clear the Emergency Stop signal, as:

```
refined clear_emergency_stop of clear_signal is
when ... r.emergency_stop
do
    if not(r.external_hold or r.command_hold or r.alarm) then
        ...
    end if;
    r.emergency_stop := false;
end;
```

but this is not a proper refinement as is defined in DisCo, although if it were possible to program, the resulting system would still be a refinement of Robot_Signal. We could not find a simpler way around this problem than writing two actions for clearing each signal:

```
refined clear_emergency_stop1 of clear_signal is
```

[2]In version 3.0 of the DisCo Language, states have been removed and replaced by enumerated values. This new feature could be used here in advantage.

```
when ... r.emergency_stop and
        not(r.external_hold or
        r.command_hold or
        r.alarm)
do
    ...
    r.emergency_stop := false;
end;

action clear_emergency_stop2 by r: Robot is
when r.emergency_stop and
        (r.external_hold or r.command_hold or r.alarm)
do
    r.emergency_stop := false;
end;
```

We found this kind of refined actions appearing quite frequently in our systems and think that it would be an interesting improvement if the DisCo language could let us define a single action for them.

Finally, let us note that of all the signals, the only one that can be cleared by Telesys is the Command Hold signal which can only be raised by Telesys itself. For simplicity, we preferred to consider both, the clearing of the Command Hold signal and the setting of the robot destination, a single action.

```
refined set_Destination is
when ... not(r.emergency_stop or r.external_hold or r.alarm)
do
    ...
    r.command_hold := false;
end;
```

Notes

As the only way available to Telesys to stop the robot is by raising a **Command Hold signal** –by means of a **HOLD ON** command sent to the **YASNAC ERC**, the guard of the imported action *stop_Robot* is strengthened to **false**. Its functionality has been replaced by that of action *hold_on*.

Refer to Appendix A (page 79) for the complete listing of system Robot_DifSig.

3.2 The Camera Subsystem

In order to reinforce security inside the robot workcell, a Smart Camera MAPP2200 was used. This video camera has a programmable unit able to perform different tasks like edge- and motion-detection. In the TELESAFE project, this camera was dedicated to check that the workcell was in an acceptable state before teleoperation starts and, once this has been started, to detect any movement inside the workcell not produced by the robot.

As checking the state of the workcell prior to teleoperation is of a static nature and we are more interested in the dynamics of the system, we have decided not to model this function of the camera and have concentrated just on the motion-detection feature.

Even though the Smart Camera is programmable, the program that enabled it to do motion-detection was provided to us "as-is" by a different working team in VTT. This implied that we had to adapt our software to the restrictions imposed by the motion-detection software loaded into the camera.

The software controlling the camera consisted of a single control loop that periodically reads the serial port for the reception of any of two commands. The first command is used to put the camera into the motion-detection state. The other one performs just the inverse function, returning the camera to the initial idle state. If the camera detects any movement while being in the motion-detection state, it sends a simple packet through the serial port and moves to its idle state.

We started with a very simple Camera system, where we made no distinction between the state change that results from motion-detection and the change that occurs when the user sends a command to set the camera idle (even when no motion has been detected). The resulting system proved to be sufficient for our specification purposes. The difference between the two state changes above is only noticed by the user, but it is unsubstantial to the camera state.

The specification of the Camera system is:

```
system Camera

is

    class Camera is
        state *idle, detecting;
    end;

    action set_camera_idle by cm: Camera is
    when true
    do
        → cm.idle;
    end;

    action start_detection by cm: Camera is
    when true
    do
        → cm.detecting;
    end;

end;
```

Note that the guards of the actions have been kept as non restricting as possible.

3.3 The Simulator Subsystem

For the **TELESAFE** project, the simulation software used at the remote site was **TELEGRIP**^{TM} (Tele-Interactive Graphics Robot Instruction Program). **TELEGRIP** is a 3D robot simulation and off-line robot programming environment developed by Deneb Robotics, Inc., USA.

TELEGRIP provides an LLTI (Low Level Teleoperation Interface) to interact with user-defined functions. From the point of view of the Teleoperation System, it is only of concern to know the state of the LLTI associated to the model (i.e. LLTI active or inactive) and, if the LLTI is active, the position of the robot according to the Telegrip model. With these in mind, we give a class definition for Telegrip:

```
class Telegrip is
    state *inactive, active( position: integer );
end;
```

Essentially, an LLTI is composed of three functions: *init, read* and *close*. *Init* is invoked whenever the LLTI is activated by the operator of **TELEGRIP**. On the other side, *close* is invoked every time the LLTI is deactivated. *Read* is called periodically (in fact, over 600 times a second, but this is hardware dependant). This poses a restriction to the amount of time during which the read function can be active.

The init and read functions can easily be modelled like this:

```
action activate_Telegrip by tt: Telegrip is
when tt.inactive
do
   → tt.active;
end;

action deactivate_Telegrip by tt: Telegrip is
when tt.active
do
   → tt.inactive;
end;
```

As the *read* function concerns the real interface between the Telegrip system and its environment, it will be treated later, during the specification of the system (Telesys, page 22). Anyway, the user of **TELEGRIP** (i.e. the remote operator, its program or even the LLTI!) can modify at will the position of the robot, independent of the execution of the *read* function. Of course, we cannot put any restrictions on the occurrence of environmental events. Being it such an event, the action responsible for the movement of the robot model has a **true** guard:

```
action change_Pos( p: integer ) by tt: Telegrip is
when tt.active and p >= 0
do
   → tt.active( p );
end;
```

Notes

Notice that we have used a position parameter in action change_Pos. This is the usual way to represent nondeterministic inputs in DisCo.

Refer to Appendix A (page 84) for the complete listing of system Telegrip.

3.4 Combining the Components in a Single System

The whole Telesys system is composed of the Robot, the Camera and the Telegrip systems, all working distributively but cooperatively. This is one of the main reasons why DisCo was chosen as the specification language for this project.

Classes

We present a Telesys class, which is a container for all state data related to the management of the communication between the different components of the system. We intend to use only one instance of this class in our model.

Let us now start analyzing the state structure of this class.

First of all, notice that it is always rather convenient to have the possibility to decide whether we want to teleoperate or not. This leads us to the realization that Telesys must always be in any of the two following states:

```
state *inoperative, operative(last_pos: integer);
```

We assume that *inoperative* is the default state, while we should leave for a moment the explanation of the integer variable *last_pos* (see page 24).

Then, noticing that there is no way to inquire the camera to know whether it is doing motion-detection or not, this information has to be maintained by the Telesys object itself. If Telesys is in its operative state, we can safely assume that the camera is doing motion-detection (we assume no possibility of Camera failure in this model. In fact, this whole specification assumes that there cannot be malfunction error in any of its components). While, on the other side, if the Telesys object is in the inoperative state, we cannot tell. Thus, we extend this last state with the necessary sub-states:

```
extend inoperative by
   state *camera_off, camera_on;
end inoperative;
```

Finally, considering that the rate at which the Robot and the simulated robot model might move can be different, we need a way to gather position coordinates that cannot be handled to the real robot because, for example, it might be currently moving. This is easily done with the introduction of a buffer of positions (integers). The complete class definition for Telesys is:

```
class TeleSys is
   state *inoperative, operative(last_pos: integer);
   extend inoperative by
      state *camera_off, camera_on;
   end inoperative;
   buf: sequence integer;
   initially buf = <>;
end;
```

Actions

As a way of attaining a conceptual organization that will be fully exploited in the next chapters, we will group the actions into three categories:

1. **Starting**: Actions that move Telesys into state *operative* and clear the buffer of positions.

   There are two ways to start the Telesys, depending on the state of the camera.

   If the camera is already doing motion-detection, to start the Telesys we just have to update the model robot position in Telegrip (to synchronize it with the real robot, which must not be in movement) and check that the **LLTI** is active. Besides that, the only signal that could be raised in the **YASNAC ERC** controller is the *Command Hold* signal, as it can later be cleared using software.

```
refined start of change_Pos by ... ts: TeleSys; r: Robot is
when ... ts.inoperative.camera_on and
        not(r.external_hold or r.emergency_stop or r.alarm) and
        r.still and p = r.position
do
   ts.buf := <>;
   → ts.operative(p);
   ...
```

end;

If the camera is not doing motion-detection, we have to combine the previous action with action start_detection from the Camera system.

combined start_with_cam **of** Telegrip.change_Pos, Camera.start_detection;
refined start_with_cam by ... ts: TeleSys; r: Robot is
when ... ts.inoperative.camera_off and
not(r.external_hold or r.emergency_stop or r.alarm) and
r.still and p = r.position
do
ts.buf := <>;
→ ts.operative(p);
...
end;

2. **Operating**: Actions modify the robot model position in Telegrip, read the new position into Telesys, transmit it to the robot and operate it.

   The action to modify the robot model position has already been defined (change_Pos) but in order to simplify the simulation in the DisCo tool, we found convenient to define the following alternative, that prevents the movement of the model before its current position has been read into the Telesys.

refined move_Model **of** change_Pos **by** ... ts: Telesys **is**
when ... (ts.inoperative or tt.active.position = ts.operative.last_pos)
do
...
end;

We read a robot model position from Telegrip into Telesys, only when the position in Telegrip is different from the last position loaded. This is the reason to keep the value of the last position read in variable *operative.last_pos*. The new position is then appended to the buffer of positions.

action get_new_pos **by** ts: Telesys; tt: Telegrip **is**
when ts.operative.last_pos /= tt.active.position
do
→ ts.operative(tt.active.position);
ts.buf := ts.buf &[3] <tt.active.position>;
end;

To transmit a position to the robot, we use action set_Destination, making sure that its parameter p equals the first element in the buffer of positions

[3] & is the operator for catenation of sequences in DisCo.

refined operate **of** set_Destination
by ... ts: TeleSys is
when ... ts.operative and
p = head(ts.buf)
do
...
ts.buf := tail(ts.buf);
end;

3. **Stopping**: Actions that move Telesys into state *inoperative*.

   The reasons why the Telesys might have to be stopped are:

   - A signal is raised in the robot controller (The possible signals are: Emergency Stop, External Hold and Alarm).

     action ack_Signal **by** r: Robot; ts: Telesys **is**
     when ts.operative and
     (r.external_hold or r.emergency_stop or r.alarm)
     do
     → ts.inoperative.camera_on;
     end;

     Notice that the raising of a signal does not produce any reaction from Telesys, besides changing the state of the Robot object. This reflects one of the restrictions imposed by the controller. As the controller cannot generate its own messages to the rest of the Telesys system, we need a separate action to do the acknowledgment of the signal by Telesys.

     Notice as well that if a signal is set and cleared before it is acknowledged, the rest of the system will not acknowledge it.

   - The camera detects movement inside the robot workcell.

     refined motion_detected **of** stop **by** ... ts: Telesys **is**
     when ... cm.detecting
     do
     ...
     → ts.inoperative.camera_off;
     end;

   - The operator of the system issues a command to stop.

     refined stop **by** ... ts: Telesys **is**
     when ... ts.operative
     do
     ...
     → ts.inoperative.camera_off;
     end;

- The LLTI in **TELEGRIP** is deactivated (*close* function called).

```
refined deact_Telegrip by ... ts: Telesys is
when ... ts.operative
do
    ...
    → ts.inoperative.camera_off;
end;
```

Notes

Notice that before its refinement, deact_Telegrip is the combined action of Robot_DifSig.hold_on, Camera.set_Camera_Idle and Telegrip.deactivate_Telegrip.

In case the LLTI in TELEGRIP is deactivated while Telesys is not operative, we have:

```
refined deactivate_Telegrip by ... ts: Telesys is
when ... ts.inoperative
do
    ...
end;
```

Some of the imported actions are no longer valid and so are permanently disabled in the Telesys system. Refer to Appendix A (page 85) for the complete listing of system Telesys.

3.5 Initial Conditions

We assume there is only one instance of each class defined, and we have given each object the name of its class. Therefore the specification only declares the following objects: Telegrip, Camera, Robot and Telesys. Though the complete initial conditions can be deduced from the initial conditions for each of the classes, we summarize them with this global DisCo assertion:

```
initially InitTelesys is
    Robot.still.idle
    and not(emergency_stop or external_hold
        or command_hold or alarm)
    and Camera.idle
    and Telegrip.inactive
    and Telesys.inoperative.camera_off and Telesys.buf = <>;
```

3.6 Fairness

3.6.1 Environmental and Implementation Actions

The specification of Telesys -as well as any other DisCo specification- is a closed one, which means that it models not only the possible implementations of the system, but it also models the behavior of the environment. A good specification makes a clear distinction between actions performed by the environment and actions performed by the implementation, as only the later have to be implemented [Mik95]; and, in order to do it, the events produced by the environment and the implementation have to be identified first.

Besides that, a good specification is also one which imposes no arbitrary and unnecessary restrictions over the environment actions. A system cannot restrain its environment, it only reacts to environmental events.

Let's take a look now at the Telesys system we have just given and try to determine which events are produced by the environment and which by the implementation: Those events like

the order to start the system, activate/deactivate the TELEGRIP LLTI, stop the system and move the model robot around in TELEGRIP, are clearly environment events. Also produced by the environment are those events like the turning-on/off of a signal at the robot controller or a modification inside the robot workcell that causes the camera to detect motion. On the other side, the actions that gets the new robot coordinates from the simulation software (get_new_pos), transport the robot coordinates to the actual robot and orders its movement (operate), and acknowledges a signal raised at the robot controller (ack_signal), are clearly the response to implementation events.

We then have implementation actions (*get_new_pos, operate* and *ack_signal*), environment actions (the various actions that turn a signal on/off, *move_Model, activate_Telegrip*) and actions that perform tasks associated in part with the environment and in part with the implementation (*start, start_with_cam, stop, deact_Telegrip* and *motion_detected*).

The problem of specifying an implementable interface in the context of DisCo has been thoroughly studied in [Mik95] and we will not meddle with it. Instead, we will keep the specification of the interface as simple as possible.

We have considered that some external events could be modelled as having an immediate effect on the system (i.e., actions *start, stop*, etc.) while for other events, because of restrictions imposed by the way the components can communicate between them, we had to split the event occurrence from the reaction generated by it (i.e., actions *ack_signal* and *get_new_pos*).

3.6.2 Fairness Properties of Implementation Actions

After identifying environment and implementation events, and noting that some actions in Telesys are both environment and implementation actions, we conclude that, at this point of the specification, we can only impose fairness restrictions over the actions *get_new_pos, operate* and *ack_signal*.

Though we are not dealing with real-time here, it seems natural to ask any sensible implementation to eventually force the execution of internal actions if they are infinitely enabled [Ks96]. But, as the set of states at which these actions are enabled are not disjoint, we will require strong fairness for each of them. Therefore, we have:

$$SF(A_{get\_new\_pos}) \wedge SF(A_{operate}) \wedge SF(A_{ack\_signal})$$

where A_x is the TLA action associated with DisCo action x.

3.6.3 Assumptions about the Environment

Though the system cannot restrain the environment in any way, we have to make some assumptions about the environment to be able to deduce some interesting properties of the system.

We have already introduced one such assumption when defining action *move_Model*, which is disabled if the system has not loaded the last position yet. In this case, we are not posing a restriction on the environment but assuming that the loading of robot positions into system will always be faster than the simulated movement of the robot in TELEGRIP. It is a realistic assumption since the calling of function read in the LLTI has a rate of approximately 600 calls per second, while the real robot cannot change directions that fast.

If we deleted this restriction from action Move_Model, our assumption about the environment could also be expressed in TLA as

$$\Box \left[< A_{move\_Model} > \Rightarrow \begin{array}{l} Telesys.inoperative \vee \\ Telegrip.active.position = Telesys.operative.last\_pos \end{array} \right] \quad (3.1)$$

Another assumption about the environment will be to suppose that the operator of the system never clears a signal from the robot controller before it has been detected by the system. If this weren't the case, it could be possible for the robot to stop in the middle of a trajectory and restart with another direction violating the "security property" (see 3.8).

For each of the signals Emergency_Stop, Command_Hold and Alarm, and for each of the two clearing actions, we assume properties like the following:

$$\Box[< A_{clear\_Emergency\_stop1} > \Rightarrow Telesys.inoperative] \tag{3.2}$$

3.7 Some Trivial Invariant Properties

We state here a few invariant properties of system Telesys that will be of use in the following chapters:

- If there is a signal at the controller, any movement in the workcell, the system operator sends the order to stop it or the corresponding TELEGRIP LLTI is deactivated, then the system goes into the *inoperative* state.

$$\begin{array}{l} < A_{set\_emergency\_stop} \vee A_{set\_external\_hold} \vee A_{set\_alarm} \vee \\ A_{motion\_detected} \vee A_{stop} \vee A_{deact\_Telegrip} > \rightsquigarrow Telesys.inoperative \end{array} \tag{3.3}$$

  Notice that we need assumption 3.2 to prove this invariant property.

- Whenever the system is in the *inoperative* state, its estimation of the state of the camera is correct.

$$Telesys.inoperative.camera\_on \Rightarrow Camera.\text{detecting} \tag{3.4}$$

$$Telesys.inoperative.camera\_off \Rightarrow Camera.idle \tag{3.5}$$

- Whenever the system is in the *inoperative* state, the robot is not moving.

$$Telesys.inoperative \Rightarrow Robot.still \tag{3.6}$$

  Note that this invariant will not be preserved by the atomicity refinement of the following chapters.

- If the system is *operative* and there is at least a robot joint value in the buffer, then the robot will finally be moved or the system be stopped (by different possible causes).

$$\begin{array}{l} Telesys.operative \wedge \\ \neg empty(Telesys.buf) \Rightarrow \end{array} \quad \Diamond \left\langle \begin{array}{c} A_{operate} \vee A_{ack\_signal} \vee A_{motion\_detected} \vee \\ \vee A_{stop} \vee A_{deact\_Telegrip} \end{array} \right\rangle \tag{3.7}$$

3.8 Security: An Important Property

Let's consider the two following situations:

1. Suppose the robot is instructed to move from its current position A to point B and then to point C, and suppose that the order of the instructions get mixed up so that the robot goes from A to C and then to B. Clearly this is a disastrous situation as the robot can destroy something in the workcell or even destroy itself by attempting such a movement.

2. The user orders the robot to move from its current position A to point B. While it is moving, the camera detects motion in the workcell and the robot is stopped in D, a point between A and B. The user then restarts the system, assuming the robot is in position B, and orders it to move to point C. Clearly the robot movement will not be as expected by the user.

Clearly, we would like to make sure no situation like these ever happens. Informally, the property that we would like Telesys to satisfy, says:

"The path followed by the robot is always a prefix of the path followed by the model since Telesys went into the operative *state".*

Note that, by invariant 3.6, if the robot is moving, then Telesys must be in the operative state, so the property enunciated must be valid whenever the robot is in motion.

To formalize it, we need the following definition:

Definition 1 *A path is a sequence of joint-value coordinates.*

We introduce two new variables, whose value is only of importance while Telesys is in state operative:

- *mpath*: The path taken by the robot model (in Telegrip) as it has been received by Telesys (by means of action get_new_pos) since the last step verifying

$$Telesys.inoperative \wedge Telesys.operative'$$

- *rpath*: The path taken by the real robot since the last step verifying the same condition as above.

It is assumed that both variables are initially empty. Notice that the actual path taken by the model in Telesys can have at most one more step than the path received by Telesys, according to assumption 3.1.

The last step is to refine the actions in Telesys so as to update these two variables (mpath and rpath).

Actions *start* and *start_with_cam* are refined with the conjunct

$$mpath' =< r.position > \wedge rpath' =< r.position >$$

action *get_new_pos* adds the new position to *mpath* with the disjunct

$$mpath' = mpath \ \& < tt.active.position >$$

and action *operate* updates *rpath* with

$$rpath' = rpath \ \& \ head(ts.buf)$$

Considering that these are then the only actions that modify the values of mpath and rpath, the validity of the following property is easily proved.

The Security Property

$$\Box\,(Telesys.operative \Rightarrow mpath = rpath \ \& \ Telesys.buf) \tag{3.8}$$

Chapter 4

Implementation Notions

4.1 Motivation

It can be argued that the specification of a system should be independent of its design. Then, from the point of view of a purist, the decisions involved in choosing an appropriate software architecture or deciding whether the system should be a concurrent or distributed one, are outside the scope of the specifier as well as of the specification language. But, in real life, the is no clear-cut frontier between specification and design, as well as design sometimes fades into implementation (for example, notice that this exactly what happens with some prototyping methodologies).

The whole development process can be viewed as continuum that starts with a very simple and abstract model of the system and its environment and then goes into adding complexity and information in successive steps, until a model is obtained that is expressive and restricted enough to be executed in the computer. This encompasses specification, design and implementation, an idea which is at the heart of the transformational methodology advocated, among many others, by the developers of DisCo.

Consistently, if a specification language can be used to support the design process, why not use it as a design language too?

Here, we investigate this possibility in the framework of action systems and a specification language based on them (DisCo). We intend to use this language in the design of our Robot Teleoperation System.

Starting from a DisCo specification of the system, we are then interested in obtaining a system with action guards and bodies that have a direct implementation using the mechanisms provided by the programming language and operating environment to be used [KsK88].

Some design considerations taken from the Robot Teleoperation System

The TeleSys specification of the previous chapter does not describe the fact that there are inherent restrictions to the exchange of information between the local and the remote sites. The local site is where the robot is located, whereas the remote one is where the simulation software is run, and from where the teleoperator instructs the robot movements.

The TELESAFE project imposed the following restriction to the system: The local and remote sites can be situated very far from each other and they must use the usual transport protocols supported by the Internet. This restriction was added to the security objectives regarding teleoperation of robots, pursued by the whole project.

As a result, a number of design prerogatives where outlined, which the designed system should adapt to:

- There should be a clear division between data to be stored locally and data to be stored remotely.

- A number of lose-less, order-preserving, point-to-point connections could be assumed as provided by the operating environment.
- An atomic implementation of each communication between the two sites is not feasible or, at least, not desired, given the necessity to attain some level of security related to the response velocity of the teleoperation system.

The design prerogatives sketched above, have motivated us to study how to extend the DisCo methodology based on successive refinements of actions systems to manage the introduction of asynchronous communication channels at the design stage.

In other words, we are interested in studying atomicity refinement in the context of reactive action systems. The object of our present analysis is therefore, the possible relations between the refined and the original systems.

If we defined an appropriate relation between the systems, some properties previously verified for the original system, could be relatively extended to the refined one, freeing thus the designer from the burden of a bigger and more complex proof.

4.2 Background definitions

4.2.1 Behaviors of a system

Given the canonical form specification

$$S \equiv (\exists x_1, \ldots, x_n : Init \wedge \Box[A] \wedge F) \tag{4.1}$$

we define some terms that will be useful in the following sections. (Notice that most of these definitions have been taken from [Ks96, Chapter 2]).

Definition 2 *Set of states of a system. An* state *is an assignment of values to state variables. The set of all states of system S, whether reachable or not, will be denoted by $States(S)$.*

Observation 3 *It is common to identify properties with the set of states that satisfy them. In that line, predicate $Init$ is both understood as the logical predicate and as the set of states*

$$\{q \in States(S) : q[\![Init]\!]\}$$

where $q[[Init]]$ denotes the evaluation of predicate $Init$ in state q.

Definition 4 *Evaluation of a state predicate. Given a state predicate (analogously state function) P and a state q, the evaluation of P in q is the expression $q[[P]]$ which results from substituting the variables of P by the values q assigns to them.*

Definition 5 *Step. A* step *of system S is a pair of states (s,t), such that $s[[A]]t$ holds, where this last expression is obtained from A by interpreting its unprimed variables x in state s, i.e. as $s[[x]]$, and primed variables x' in state t, i.e. as $t[[x']]$.*

Definition 6 *Behavior. A behavior for system S is an infinite sequence of elements of $States(S)$ that satisfies (4.1).*

Definition 7 *Set of behaviors of a system. From now on, the set of all behaviors of systems S will be denoted by $beh(S)$.*

It is sometimes useful to be able to look at a behavior of a system without considering its stuttering steps. Given a behavior σ, we consider the sequence that results of eliminating all stuttering steps from σ. If this sequence is finite, we make it infinite by repeating the final state indefinitely. The resulting sequence is called $\widehat{\sigma}$.

4.2.2 Visible variables and locations

Distributed systems represent naturally those systems where data items are spread among a variety of locations and there is an associated and, usually substantial, delay in the transportation of data from one location to another for processing purposes.

A location is a set of visible variables that are thought of as being close enough to each other so that every transmission of data between any two of them can be modelled as an atomic transition.

Definition 8 Visible variables of a system. *These variables characterize the externally visible state of the system. Visible variables are those that are not abstracted away by quantification in the TLA expression defining the system. We denote by V_S the set of all visible variables in system S.*

Definition 9 Location. *Given the system S, and the set V_S of visible variables in S, a location is just a subset of V_S.*

Definition 10 Set of locations of a system. *Given a system S, a set of locations for system S is a finite partition of V_S.*

4.2.3 Projection of a behavior (relative to a set of variables)

First, we give an intuitive definition based on behaviors as sequences of states. Given a set l of visible variables for system S, and given a behavior σ of S such that

$$\sigma = \langle (v_0, h_0), (v_1, h_1), (v_2, h_2), \ldots \rangle$$

where the v_i are the visible states (composed of the values of all visible variables V_S) and the h_i are the hidden states (values corresponding to all variables not in V_S); then we define the *projection of σ w.r.t. l* as

$$\sigma_l = \langle v_0/_l, v_1/_l, v_2/_l, \ldots \rangle$$

where each $v_i/_l$ is the restriction of the state v_i to the variables in set l.

We are now in position of giving a couple of useful notation definitions.

Definition 11 *Projected behaviors of a system w.r.t. a set of visible variables. Given a system S, a set W of sequences of states of S and a set $l \in V_S$, the projection of W w.r.t. l is defined as*

$$W_l = \{\widehat{\sigma_l} | \sigma \in W)\}$$

Notice that we have abstracted away stuttering steps.
When W is the set of all behaviors allowed by system S, we have

$$beh_l(S) = (beh(S))_l = \{\widehat{\sigma_l} | \sigma \in beh(S))\}$$

This process of projection can be expressed in TLA by quantification of program variables. That is, the behaviors in $beh_l(S)$ satisfy the TLA formula

$$\exists x_1, x_2, \ldots, x_n : S$$

where $\{x_1, x_2, \ldots, x_n\} = V_S \setminus l$, is the set of visible variables of S that do not belong to the location l [Ks96, Chapter 2].

Observation 12 *Given a system S with visible variables V_S, $beh_{V_S}(S)$ is the set of externally visible behaviors of S after abstracting away stuttering steps.*

4.3 The Usual Definition of Implementation

The transformational approach to the development of system specifications is based on the possibility of defining some kind of relation between a higher-level specification and lower-level one. A relation commonly used is the *"implementation relation"* by which the lower-level specification is considered to be an implementation of the higher-level specification.

As was stated in [AL91, page 3], "a specification S implements a specification T if every externally visible behavior allowed by S is also allowed by T. To prove that S correctly implements T, it is enough to show that if S allows the behavior

$$\langle (v_0, h_0), (v_1, h_1), (v_2, h_2), \ldots \rangle$$

where the v_i are the visible states (composed of the values of all visible variables V_S) and the h_i are the hidden states (values corresponding to all variables not in V_S); then there exist hidden states h'_i such that T allows the behavior

$$\langle (v_0, h'_0), (v_1, h'_1), (v_2, h'_2), \ldots \rangle$$

Observation 13 *It is not necessary that the variables of S coincide with the variables of T, but it is required that the visible variables of both systems be the same, i.e. $V_S = V_T$ [AL91].*

Observation 14 *With the notations defined above, the* "implementation relation" *can be expressed like this*

$$beh_{V_S}(S) \subseteq beh_{V_S}(T)$$

The *"implementation relation"* between two systems (the specification or abstract system S and the implementation system S) can be expressed shortly and elegantly in TLA. If the systems S and T are represented using canonical TLA expressions, S is a correct implementation of T, if $S \Rightarrow T$ [Ks96, Chapter 5].

The "implementation relation" is just a relation between two specifications. Its importance is related to the fact that any real system satisfying the implementation (lower-level) specification will satisfy the abstract (higher-level) specification. Thus, some properties of the real system could be proved over the abstract, and supposedly simpler, specification instead of using the implementation specification.

Of course, not all properties of the implementation specification will be satisfied by the abstract one. In essence, the "implementation relation" preserves properties that involve externally visible variables (which are the same for both specifications) and are insensible to stuttering.

4.4 Refinement Mappings and other Simulation Techniques

To prove that there is an "implementation relation" between a lower-level specification S and a higher-level specification T, several different techniques have been developed. These techniques reduce the proof of properties of complete behaviors to reasoning about single states and state-transitions.

One of the simplest is the refinement mapping technique [Lam83] which consists in finding a function f from the states of S to the states of T such that:

- f maps externally visible variables of S identically to the corresponding visible variables of T. Remember that in trying to prove the "implementation relationship", we have assumed $V_S = V_T$.
- f maps initial states of S to initial states of T (i.e., $f(Init_S) \subseteq Init_T$)
- f maps each possible step of S to a step of T (which might be a stuttering step).

- f preserves liveness properties. (This requirement is usually relaxed leaving liveness properties to a separate treatment).

It has been long known that if there is such a refinement mapping between S and T, then there is an "implementation relation" between them [AL91, section 2.4]. However, this technique is not complete in the sense that there might be a pair of specifications S and T for which no refinement mapping can be defined.

The main result of [AL91, Completeness Theorem] states that under some restrictions over S (machine-closed) and T (internal continuity and finite invisible nondeterminism) there exists a specification S' that satisfies:

- S' is obtained from S by first adding a *history variable* (a variable which records past events) and then a *prophecy variable* (one which performs nondeterministic choice in advance).
- there exists a refinement mapping from S' to T.

There are several other simulation techniques that intend to prove an "implementation relation" between two specifications. Lynch and Vaandrager [LV94] have made a unified presentation of several simulation techniques for verification of concurrent systems, in terms of a simple untimed automaton model. These techniques include: refinements, forward and backward simulations, hybrid forward-backward simulations and history and prophecy relations (These last ones being a generalization of the history and prophecy variables of Abadi and Lamport [AL91]).

In [LV94], the authors followed an action-based approach, according to which the behavior of a system is a sequence of (visible) actions. Here, instead, we will follow the somewhat equivalent state-based approach. By the way, [AL91] also follows the latter approach . For a discussion of the relations between these two concepts, see [DNV90].

However, it is not the purpose of this work to go into the small details of these simulation techniques. We refer the interested reader to the literature for that. And recall for the following sections the importance of the result by Abadi and Lamport regarding the existence of refinement mappings.

4.5 Locations, Observers and Relativity Theory

Let's concentrate on the definition of "implementation relation" (4.3) which will be our starting point. What is the meaning of "externally visible behavior"? Essentially, it implies the existence of an external observer which, at every moment, is able to capture in a kind of snap-shot, the state of all visible variables in the system.

If we are considering a distributed system, it is natural to question the mere existence of such an observer.

One of the definitive discoveries that Physics owes to the special theory of relativity is the inexistence of simultaneity among distant events [Ein49]. To make this clearer, we retort to Bertrand Russell's argumentation: the wireless telegram travels at the speed of light, so there could be nothing faster. Whatever a person does as a result of receiving a radiomessage, it is done *after* the message has been sent. However, everything that she does while the message is travelling cannot be affected by the sending of the message. And, similarly, whatever she does cannot affect the sender before some time has passed after the emission of the message.

In other words, if two bodies are separated by a considerable distance, the first cannot affect the second but after a time interval.

Returning to our argumentation regarding distributed systems, and always according to relativity theory, there cannot be such an observer, because it would imply the transmission of information at an unlimited speed. Is this a trivial observation? We think it is not. Just consider the time it takes for any satellite communication to cross the world. The delays are no longer negligible compared to the minimum reaction time for the system.

What would happen then if we deny the existence of a global observer capable of determining "the state of all visible variables in the system"? Certainly, we must change our concept of implementation. It is necessary then to overcome the distinction between time and space based in the conviction that it is possible to describe the universe in purely spatial terms, at a given instant.

4.6 The Need for a Different Kind of Relation

Starting from the Telesys specification that we have developed in the previous chapter, we would like to transform it into a specification reflecting the fact that it is a distributed system. The specification that we are looking for should be related to the former one, but we have seen that the "implementation relation" is not appropriate for the kind of transformations we would like to apply here. As a consequence, the action refinement features of DisCo will not be of help, as these features are based mainly on the ability to induce an "implementation relation" without the need of a proof. Therefore a different relation between the higher-level and the lower-level specifications in a transformation is needed. Here, we analyze what we want of this relation and leave for the next section the study of a recent proposal in that direction.

The "implementation relation" only lets the lower-level specification perform changes to the visible variables in the same way as they are allowed by the higher-level specification. Any action refinement that would lead to splitting one or several of those changes to visible variables will result, therefore, in a lower-level specification for which the "implementation relation" does not hold.

The previous argumentation showed why we cannot pretend to use the "implementation relation" if we are specifying a distributed system and our high-level specification pays no attention to locations. The Robot Teleoperation System, Telesys, is indeed a distributed system. And its specification, as developed in the previous chapter, does not distinguish between the information generated at the robot end (the robot workcell and its neighborhood) from the information generated at the simulation end.

The DisCo Method provides a series of methodological directives to guide the construction of DisCo systems. As such, it is mainly based on the capabilities of the language to define refinements of actions and object extension. Actions are refined by strengthening their guards (by adding conjuncts) and by adding statements to their bodies (provided these statements only modify variables defined in the same system, not in an imported one). Objects are extended by the addition of variables and the extension of their state machines. Both of these constructs guarantee that the refined system satisfies an "implementation relation" with the original system. Under the light of what has been said, we conclude that the DisCo Method provides no guidance for the kind of transformation we would like to operate on Telesys. We also conclude that we cannot base that transformation on action refinements and object extensions. Instead, we need an extended methodology and a different relation between the specifications.

What we are looking for then is the possibility to define other relationships between a higher-level specification and a lower-level one, the latter been the result of the splitting of actions and the redistribution of information according to a precise definition of the locations of interest. The purpose of all this is, simply, to have a way to relate the lower-level specification to the higher-level one, as to see the latter as a more abstract view of the system. Understanding the relation between these two systems will allow us to reduce the proof of some properties of the more concrete specification to those of the abstraction.

From a general view point, we want:

1. to construct a specification of a distributed system where the difference among locations is abstracted away,

2. to write a specification at a lower level of detail where no two variables, belonging to different locations, are modified simultaneously, and

3. to be able to establish a relation between the two specifications in such a way that will allow us to *extend* some of the properties of the first to the second.

Several approaches have been taken in this direction. In the following section we analyze one of them which is based on a relaxation of the conditions imposed upon refinement mappings to establish a functional relation between two specifications.

4.7 Aggregation of Distributed Transactions

Quite recently, S. Park and D. L. Dill [PD96] proposed a method for the verification of protocols by Aggregation of Distributed Transactions. In the present section, we first give a brief presentation of the method and analyze its advantages regarding the transformation of specifications by splitting of actions. Then we shortly discuss what it means to prove properties of the specifications by means of this method and finally we give some reasons why we consider this method not to be appropriate for the kind of transformation we are looking for in this work.

For the present section we will assume the existence of two systems:

A higher-level, more abstract system

$$T \equiv (\exists y_1, \ldots, y_m : Init_T \wedge \Box[A_T] \wedge F_T)$$

and a lower-level, more concrete system

$$S \equiv (\exists x_1, \ldots, x_n : Init_S \wedge \Box[A_S] \wedge F_S)$$

System S is supposed to be the result of action splitting and data redistribution (according to a given choice of locations) being applied to system T.

4.7.1 The Abstraction Function

The method presented in [PD96] defines an abstraction function between S and T, in order to establish a functional relationship between its behaviors. The abstraction function *abs* maps lower-level states into higher-level ones

$$abs : States(S) \rightarrow States(T)$$

and must satisfy:

1. *abs* maps initial states of S to initial states of T:

$$abs(Init_S) \subseteq Init_T$$

2. *abs* maps each possible step of S to a step of T (which might be a stuttering step)

$$\forall q, q' \in States(S) : q[\![A_S]\!]q' \Rightarrow (abs(q)\ [\![A_T]\!]\ abs(q')) \vee abs(q) = abs(q') \tag{4.2}$$

Observation 15 *If we use the standard notation for including stuttering steps*

$$[A]_U = A \vee (U' = U)$$

where $U' = U$ is a shorthand for $\forall x \in U : x' = x$, then (4.2) can be rewritten as

$$\forall q, q' \in States(S) : q[\![A_S]\!]q' \Rightarrow abs(q)[\![[A_T]_U]\!]abs(q') \tag{4.3}$$

where we take U to be the set of all variables.

Observation 16 *If there is an invariant Inv that we know to be satisfied by every reachable state of system S, then we can replace (4.2) by*

$$\forall q, q' \in States(S) : (q[\![Inv]\!] \wedge q[\![A_S]\!]q') \Rightarrow abs(q)[\![[A_T]_U]\!]abs(q')$$

4.7.2 The Aggregation Method

The authors of [PD96] presented a methodology to establish a relationship between systems S and T based on the definition of an appropriate abstraction function.

To be applicable, the method requires that:

1. The state variables of system S should be divided into *specification variables* and *implementation variables*. And the state variables of system T should be exactly those *specification variables* of system S.

2. There should be a set of *transactions* which are atomic at system T, but have a non-atomic counterpart at system S. That means that each single-action transaction of T should be implemented by a set of actions of S.

3. For each transaction of S there should be an identifiable *commit-point*, that is an identifiable action that first modifies a specification variable.

The commit point is then an action that moves system S from a pre-commit state to a post-commit state. A state where every committed transaction has completed is called a *clean* state.

The method of Aggregation of Distributed Transactions consists then in the definition of an abstraction function (see section 4.7.1) such that, for every state q, it completes any transaction for which q is a post-commit state, as if the transaction had been executed completely.

4.7.3 Advantages of the Aggregation Method

The Aggregation Method has a number of interesting features. As its authors state in [PD96], the idea of abstraction functions has been used many times before to relate implementation and specification state graphs. But their method generalizes previous work in a couple of directions:

1. The method is a generalization of refinement mappings [LLOR96][AL91] in the sense that the Aggregation Method allows the merging of steps even when specification variables change more than once. As we have seen from the definition, an abstraction function is a relaxed type of refinement mapping, where the condition on the preservation of local variables has been dropped. In other words, this method accounts for the splitting of action atomicity that we are interested in here.

2. The method aggregates steps across distributed components whereas some previous work, for example [Lam83], only joined sequential steps pertaining to the same local process. As we are interested in partitioning the actions of the Telesys system according to the spatial distribution of data, we will certainly have to split actions into sub-actions modifying different components of the distributed system.

However convenient these features of the method could be, we maintain that it can be of little use before some points are properly clarified. The presentation of the Aggregation Method in [PD96] does not formalize on the criteria for the election of a proper abstraction function and thus, it is not clear what kind of relationship is established between systems S and T. The authors even wrote that they have used the method to prove the cache coherence protocol for the FLASH microprocessor, but the paper does not include any means as to relate proofs of the implementation to proofs of the specification.

4.7.4 Proving Properties

Here, we will analyze the Aggregation Method from the point of view of the relation it establishes between systems. The next proposition gives a way to relate properties of the higher-level system to those of the lower-level one. But first, we need a definition:

Definition 17 *Given* $\sigma =< q_0, q_1, \ldots >\in beh(S)$, $abs(\sigma)$ *is the behavior of* T *that results from applying function* abs *to every state* q_i

$$abs(\sigma) =< abs(q_0), abs(q_1), \ldots >\in beh(T)$$

Proposition 18 *If* T *satisfies property* W, *then* S *satisfies property* $abs^{-1}(W)$.

Proof. Suppose that T satisfies property W, then we know that $beh(T) \subseteq W$ and as abs is an abstraction function, $abs(beh(S)) \subseteq beh(T)$. Therefore $abs(beh(S)) \subseteq W$.

By applying the inverse of abs, and noticing that $beh(S) \subseteq abs^{-1}(abs(beh(S)))$, we get

$$beh(S) \subseteq abs^{-1}(W)$$

which means that system S satisfies property $abs^{-1}(W)$.

□

Observation 19 *To prove that a lower-level specification* S *satisfies property* P, *by means of the previous proposition, it is enough to find a property* W *of the higher-level specification* T *such that* $abs^{-1}(W) \subseteq P$.

Certainly, the complexities of finding such a property W will vary depending on the properties of the abstraction function itself. Let's consider two simple cases as to clarify the matter a bit:

1. Suppose that q_t is an initial state for system T. In other words,

   $$q_t \in Init_T$$

   and let's define $abs : States(S) \rightarrow States(T)$ as

   $$abs(q) = q_t$$

   for every $q \in States(S)$.

   Obviously, this choice of abstraction function will be of little help in proving any property, as $abs^{-1}(W)$ is the empty set if $q_t \notin W$, or the whole of $States(S)$ otherwise. By the way, this is why we affirm that the authors of [PD96] have not given criteria to choose an appropriate abstraction function.

2. Suppose now that we know abs to be a refinement mapping. That means that the sets of visible variables for both systems, S and T, coincide (i.e. $V_S = V_T$) and that $abs/_{V_S} = id_{V_S}$ (the abstraction function behaves like the identity over visible variables). Suppose then that property P is an assertion over the values of V_S and that we wanted to prove that system S satisfies P. That is, we have to prove that

   $$beh_{V_S}(S) \subseteq P$$

   Knowing that abs behaves like the identity for the visible variables, we have $abs^{-1}(P) = P$. Therefore, by the observation above, it suffices to prove that system T satisfies property P, or that

   $$beh_{V_T}(T) \subseteq P$$

These two cases just considered show that the election of the abstraction function is critical to the kind of properties that can be translated from the higher-level specification to the lower-level one. They also show that the more we know about the definition of the abstraction function, the richer properties we can prove.

4.7.5 A Critique

The following is an enumeration of the difficulties and problems we encountered while trying to apply the Aggregation Method to the Robot Teleoperation System.

- The definition of the abstraction function serves the sole purpose of determining a relation between the specifications. That is, the conditions posed over the abstraction function only guarantee that a state to state relationship (expressed functionally in this case) can be extended to whole behaviors.
- The properties that can be proved for the lower-level specification by reducing them to properties of the higher-level specification using the method depend heavily on the choice of the relation between abstraction and implementation chosen.
- The method requires a precise definition of the abstraction function, at least for reachable states. This could be time consuming and should be only attempted after deciding which properties to prove.
- But, there is no clear way to relate the properties that the abstraction function must fulfill, to the properties we want to prove for the lower-level specification.
- There is the idea that in any clean state, the values of the specification variables in S are equal to those of the variables of T, but the method does not formalize it enough.
- The method is too restrictive as long as it requires that the elements of $States(S)$ possess enough information as to reconstruct the states of system T. Therefore it will not be always possible to define the abstraction function.

  As was said in section 4.4, we can always enrich the states of system S with history and prophecy variables that could help in defining the abstraction function. Anyway, that means changing system S.

Even if the Aggregation Method can account for the splitting of actions and the aggregation of steps across distributed components, it is not appropriate for the kind of transformation we would like to apply to the Telesys specification of the previous chapter. The reasons for this are:

- *The method, as presented in [PD96], requires the specification variables of system S to be the same variables as of system T.*

  As we want a specification of the Teleoperation System that takes into account the local and remote locations, we cannot pretend that every variable will be destined to a unique location. The information held in some of the variables will have to be redistributed. For example, variable Telesys.buf holds the information in transit between the local and the remote locations. Therefore, some of this information will be at a moment in one location and some other information at the other location. This means that we will have to split not only actions but also variables, something which is not allowed, at least directly, by the Aggregation Method.

- *There is no provision for the interruption of the execution of a transaction by the execution of other transaction.*

  The Aggregation Method suggests that we commit a transaction as soon as a specification variable is first changed. This rules out any possibility of cancelling a transaction even when it has already changed a specification variable.

  In our Teleoperation System, we cannot lock any part of it just because a transaction is waiting for a message to get through. It is simply unreasonable not to attend any emergency situation just because a message is taking some time to go from one component to the other. To cope for these situations, we need a way to define the possible interruption of a transaction by the execution of another.

In the next section, we present a method that attempts to overcome this problems and difficulties.

4.8 Synchronized Combination

In the previous section we have argued that the Aggregation Method cannot cope with the kind of transformation we would like to operate on the Telesys specification.

It is worth noticing that we have found that the Aggregation Method has been applied long before –under different names– by many researchers (including those working with the DisCo Language [Mik95][Ks95]) and also in somewhat more general ways. Taking into account these generalizations, we present here a description of the method to be used in the transformation of the Telesys system developed in the previous chapter.

We are not pretending this method to be new. In fact, we have adopted the name "Synchronized Combination" from [Ks95] where it was used to prove the correctness of a solution to "The RPC-Memory Specification Problem" [BL94]. In [Mik95], a similar method was used to prove the refinement of an interface in DisCo. Our contribution can be found in the presentation of this method immersed in a framework where its relationship with refinement mappings and abstraction functions is evident. We expect to provide a sufficiently general view so as to ease the application of the method to very diverse cases.

4.8.1 Motivation

From the discussion in the previous section, we foresaw two possible ways to generalize the aggregation method (remember that S is our low-level concrete system whereas T is the high-level abstract one):

1. The abstraction function could be replaced by an unrestrained binary relation.

2. Instead of adding history and prophecy variables to system S, a complete copy of the variables in system T could be added to S (thus producing a "combined specification" of S and T).

The first generalization results from appreciating that, in proving properties of system S, it is necessary to be able to calculate the inverse of the abstraction function (see section 4.7.4). In general, the inverse is not a function, just a binary relation. In that case, we seem to loose nothing in replacing the abstraction function by a binary abstraction relation.

The purpose of the second generalization is twofold: First, it fixes a deficiency in the aggregation method as the state of the "combined specification" now holds enough information to reconstruct the state of T. And second, it lets us define the abstraction relation as a relation between the elements of the state of a single system. Moreover, this last observation will allow us to give an implicit definition of the abstraction relation –rather than the explicit relation required by the Aggregation Method.

The main idea of the method consists in synchronizing each atomic transaction of the higher-level system T, with one of the steps belonging to the same transaction, but in S. This will establish the implicit abstract relation

4.8.2 The Synchronized Combination Method

The main objective of the Synchronized Combination method is the implicit definition of a relation R between the states of systems S and T, such that every state of S (the implementation) is included in the relation.

We have organized the method by dividing it into three interrelated phases:

1. **Combination Phase**

   This phase consists in the definition of a synchronized system $SYNC(S,T)$ out of systems S and T. The definition of $SYNC(S,T)$ provides the implicit definition of the relation R between the states of S and T. Notice that relation R replaces the abstraction function *abs* from the Aggregation Method.

2. **Verification Phase**

   By proving that $SYNC(S,T)$ satisfies a given precise restriction, this phase verifies that every behavior of system S can be extended to a behavior of system $SYNC(S,T)$. Notice again that this is the counterpart of property 4.2, which was required for the abstraction function *abs* according to the Aggregation Method.

3. **Property Translation Phase**

   Once relation R has been defined, it is used to relate a (not yet proved) property of system S, to one or several (already proven) properties of system T. Notice that though this phase depends on the previous two, the way the $SYNC(S,T)$ system is constructed should be also determined by the kind of properties that one would like to prove by using this method.

We turn now to describe each phase in detail.

Combination Phase

As before, we will assume the existence of two systems:
A higher-level, more abstract system

$$T \equiv Init_T \wedge \Box[A_T]_{Vars(T)} \wedge F_T$$

and a lower-level, more concrete system

$$S \equiv Init_S \wedge \Box[A_S]_{Vars(S)} \wedge F_S$$

System S is supposed to be the result of action splitting and data redistribution (according to a given choice of locations) being applied to system T. We also require:

- $Vars(S) \cap Vars(T) = \emptyset$ (if not, the name clash is resolved by renaming one of the colliding variables).
- The set of actions of system T (idem S) is $Actions(T)$ (resp. $Actions(S)$). Each action is assumed to be distinguishable from the others.

The synchronization of both systems consists mainly in assigning, to every action in system S, a set of actions taken from system T or a stuttering step. Then, if $\overline{P}(X)$ denotes the set of non-empty subsets of X, we are looking for a function

$$g : Actions(S) \rightarrow \bar{P}(Actions(T) \cup \{Unchanged_{Vars(T)}\})$$

where $Unchanged_{Vars(T)}$ is a property satisfied by any stuttering step of system T.

Definition 20 *We are now in a position to define the synchronized system (Notice that, as we are not interested in liveness properties here, we do not take into consideration F_L nor F_S):*

$$SYNC(S,T) \equiv Init_{SYNC} \wedge \Box[A_{SYNC}]_{Vars(SYNC(S,T))}$$

where

$$\begin{array}{rcl} Init_{SYNC} & \equiv & Init_S \wedge Init_T \wedge R^* \\ Vars(SYNC(S,T)) & = & Vars(S) \cup Vars(T) \\ A_{SYNC} & = & SYNC_1 \vee \ldots \vee SYNC_n \\ n & = & |Actions(S)| \end{array}$$

and, for every i, such that $1 \leq i \leq n$,

$$SYNC_i \equiv S_i \wedge (B_1 \vee \ldots \vee B_k) \wedge (R^*)'$$

assuming that $g(S_i) = \{B_1, \ldots, B_k\}$.

SYNC(S,T) is well-defined. To guarantee that the actions of the synchronized system are correctly defined, it has to be proved that, for every action $SYNC_i \in Actions(SYNC(S,T))$,

$$\forall u \in States(SYNC(S,T)) : u[[Enabled(SYNC_i)]] \Rightarrow (\exists v : u[[SYNC_i]]v)$$

Observation 21 *R^* is an expression containing only unprimed variables. It is the designers responsibility to define R^* appropriately so as to induce the desired relation R between systems S and T. Notice as well that R^* constitutes an invariant for system $SYNC(S,T)$.*

Observation 22 *From the definition it can be easily proved that system $SYNC(S,T)$ satisfies an "implementation relationship" with respect to each one of systems S and T. Then, according to observation 14*

$$beh_{Vars(S)}(SYNC(S,T)) \subseteq beh(S) \tag{4.4}$$

and

$$beh_{Vars(T)}(SYNC(S,T)) \subseteq beh(T)$$

Verification Phase

Once system $SYNC(S,T)$ has been defined, it induces a relation R between systems S and T in the following way:

Definition 23 *Implicit relation R. Given $(e_s, e_t) \in States(S) \times States(T)$, then $e_s R e_t$ if and only if there exists a behavior $\sigma =< \sigma_1, \sigma_2, \ldots > \in beh(SYNC(S,T))$ and an index i, such that $\sigma_i = (e_s, e_t)$.*

This phase then consists in verifying that each possible behavior $\tau \in beh(S)$ can be extended to a behavior $\sigma \in beh(SYNC(S,T))$ such that $\sigma_{Vars(S)} = \tau$. In other words, in this phase we try to prove that system $SYNC(S,T)$ satisfies the following restriction:

$$beh(S) \subseteq beh_{Vars(S)}(SYNC(S,T)) \tag{4.5}$$

Observation 24 *If restriction 4.5 is proved to hold, then by 4.4, we know that*

$$beh_{Vars(S)}(SYNC(S,T)) = beh(S)$$

This equality indicates that if we wanted to prove a property for system S, we would just have to prove it for system $(\exists x_1, \ldots, x_n : SYNC(S,T))$, where $Vars(S) = \{x_1, \ldots, x_n\}$.

Finally, in order to prove 4.5, it suffices to verify that:

1. $\forall e_s \in States(S) : e_s[[Init_S]] \Rightarrow (\exists e_t \in States(T) : e_t[[Init_T]] \wedge e_s R^* e_t)$
2. System $SYNC(S,T)$ satisfies

$$\Box(Enabled(S_i) \Rightarrow Enabled(SYNC_i))$$

for every action $S_i \in Actions(S)$.

Property Translation Phase

During this phase, a property W that is to be proved for system S, is translated into a property $\widetilde{W}$ of system T by means of the relation R. It is assumed that proving $\widetilde{W}$ is considerably easier than proving W and that R guarantees that W holds whenever $\widetilde{W}$ holds.

It is worth noticing that we would need to make partially explicit the characteristics of relation R to be able to use it in the proof, and that not every property W can be proved using this method. Also notice that there are two points where the designer can affect R, namely: The choice of function g which determines how actions are tied together; and the choice of relation R^*, which helps in adjusting the synchronization whenever an action of system S is synchronized with more than an action of S or whenever there are action parameters.

Since this phase depends absolutely on the choice of property W and relation R, we refer the reader to section 6.3 for an example of the application of this phase to the Robot Teleoperation System.

The following chapters describe our experiences in applying the **Synchronized Combination** method to the Robot Teleoperation System.

First, in chapter 5, a lower-level specification of the Robot Teleoperation System is presented. This specification, called *TeleSysLoc*, takes into account the existence of a local and a remote location.

Then, chapter 6 describes each of the three phases of the method as they were applied to *TeleSysLoc*. Section 6.1, which corresponds to the *Combination Phase*, defines system $SYNC(TeleSysLoc, Telesys)$. Section 6.2, corresponding to the *Verification Phase*, makes sure that the synchronized system defines an adequate relation between *TeleSysLoc* and *Telesys*. Finally, section 6.3 shows that *TeleSysLoc* satisfies an adapted version of the **Security Property** by reducing the proof to that in *Telesys* (see section 3.8).

Chapter 5

Partitioning TeleSys

5.1 Motivation

The DisCo specification of system Telesys that was given in section 3 has the problem, from the designer's point of view, that it does not reflect the underlying communication architecture used in the implementation. In this chapter, by introducing locations as defined in page 33 (definition 9), we take a first step towards the inclusion of these design characteristics in the specification of the system.

Telesys is a teleoperation system meant to connect a robot with a simulator possibly thousands of kilometers apart. Due to the current technology and relativistic restrictions, it seems unreasonable to pretend atomicity for the actions supposed to act on both ends of the system (robot and simulator). There are security reasons as well for not implementing some actions as atomic: In the event of an emergency, the robot must be stopped as soon as possible and this cannot be prevented by any locking mechanism (or of other kind) guaranteeing the atomicity of actions that involve the communication of the more distant components of the system.

The choice of locations is a rather arbitrary one and there are many possibilities. For instance, we could choose to place the Robot and the Camera in different locations. Instead, we have based our election of locations in the physical distance between the components, obtaining only two different locations:

- *Local* location: Includes the Robot and the Camera.
- *Remote* location: Includes the simulation software (Telegrip).

Notice that Telesys, being involved with the communication and coordination of the three components above (Robot, Camera and Telegrip), should be split into two parts, each one for a different location (More on this later).

5.2 Requisites of the Partitioned System

Having defined our set of locations (see page 33) , we intend to build a new system which we will call TeleSysLoc. This system will be based on Telesys, and we intend it to fulfill the following requisites:

1. *No* action in the system can modify more than a *single* location.
2. The events produced by the environment are modelled as actions modifying *only* the location where the event occurs.
3. There is some formal way of relating properties of the original system (i.e., Telesys) with properties of the partitioned system (i.e., TeleSysLoc).

Note that by the first requisite, TeleSysLoc will induce implementations that do not need any implicit locking mechanism for any communication involving the two separated locations (Local and Remote). The second requisite prevents any implicit communication between the location where the event occurs and where the system reaction to it initiates. The last requisite points to considering system Telesys as an abstraction over TeleSysLoc, and thus, to considering Telesys as a guide into the construction of the more concrete system.

5.3 Constructing the Partitioned System

We proceed now to construct system TeleSysLoc, starting from Telesys and trying to satisfy the requisites described above. We consider that the construction procedure used in this section can be used as a guide in solving similar tasks, where a DisCo system has to be partitioned according to a given set of locations.

5.3.1 Assigning Variables to Locations

For each piece of information, we try to determine a location where it belongs.

For those DisCo objects clearly associated with a location, the decision is trivial. In that way, all variables belonging to the Telegrip object belongs to the Remote location while those variables in the Robot and Camera objects belong to the Local location. Moreover, as these systems will not be modified at all in this transformation, once we have assigned a location to any of them, we do not need to care about the locations of any variables added to them.

Partitioning the Telesys Object The situation with the variables in the Telesys object is not as simple.

A Telesys object has essentially two states: inoperative and operative. There are actions that operate the transition between these states that, coincidentally, modify both locations. As we intend to replace each these actions by a set of actions respecting requisite 1 above, we will add a state to represent a moment in the middle of a transition. As there are only two states in Telesys, we need two new states, one for what we have called "the starting process" (i.e., going from inoperative to operative) and the other for "the stopping process" (i.e., going from operative to inoperative).

Consequently, we start with the following state description:

state *NOT_OP, STARTING, STOPPING, OPER;

where state NOT_OP corresponds to Telesys.inoperative, OPER to Telesys.operative and the other two, represent each of the transition processes, as described.

Finally, notice that, in a way, a Telesys object controls the two ends of the communication between the locations. We have chosen to replicate its state machine in two new objects: R_Telesys (the part of Telesys situated in the Remote Location) and L_Telesys (situated in the Local location).

There were two reasons for using a buffer of positions in the Telesys object: buffering the arrival of data from Telegrip and buffering the sending of data to the Robot. As each of these activities should be performed at a different location, we use a buffer in the two new objects. Up to now, we have:

```
class R_TeleSys is
    state *NOT_OP, STARTING, STOPPING, OPER;
    buf: sequence integer := <>;
end;
```

Class L_Telesys is defined similarly.

We will pretend that the system is operative if both, R_Telesys and L_Telesys are in their OPER states; while we will say the system is inoperative if they are in their respective NOT_OP

states. In the forthcoming sections, we will have to give some meaning to ambiguous situations like R_Telesys.STARTING and L_Telesys.STOPPING and we will also have to prove that some combinations of states will never be reached.

5.3.2 Determining the Location of Environmental Events

To be able to fulfill Requisite 2, we first have to determine which is the location were an environmental event is first detected. As we are assuming there can be no simultaneous events, it is sensible to pretend this determination to be possible.

The environmental events in Telesys are:

1. Events produced at location Remote:
   (a) The Telegrip LLTI is activated/deactivated (actions activate_Telegrip and deact_Telegrip).
   (b) The robot position is changed in the simulator (action move_Model).
   (c) The remote operator asks the system to start teleoperation (actions start and start_with_cam).
   (d) The remote operator asks the system to stop (action stop).
2. Events produced at location Local:
   (a) A signal is raised at the robot controller (actions set_emergency_stop, set_external_hold, etc.)
   (b) A signal is cleared at the robot controller (actions clear_alarm1, clear_alarm2, etc.).
   (c) Motion is detected by the camera inside the workcell (action motion).
   (d) The robot reaches its destination (action move_Robot).

Of all of the above, 1.3, 1.4 and 2.3 are the only events with associated actions that modify more than one location.

5.3.3 Defining Reactions to Environmental Events

Event 1.1 Activation/Deactivation of Telegrip We only have to consider what to do when Telegrip is deactivated, because the activation is handled properly by class Telegrip. Basically, the deactivation of the Telegrip LLTI forces the system to follow the "stopping process" if it is already operating or in the "starting process". As this is a Remote event, the reaction can only modify variables in the Remote location:

```
refined deactivate_Telegrip by ... rts: R_Telesys is
when ...
do
   ...
   if (rts.OPER or rts.STARTING) then
      → rts.STOPPING;
   end if;
end;
```

Notice that the "stopping process" has priority over the "starting process".

Event 1.2 Movement of the Model Robot As was said in page 27 (Assumptions about the Environment), we have introduced some assumptions about the environment events in the guards of the reactions to them, noting that we don't pretend to restrict the environment, but to reflect its properties in the system.

After changing the state machine associated with the remote copy of the Telesys object (i.e., R_Telesys), we have to change accordingly the definition of the actions:

```
refined move_Model of change_Pos by ... rts: R_Telesys is
when ... (not(rts.OPER) or tt.active.position = rts.OPER.last_pos)
do
   ...
end;
```

Event 1.3 Start Request The "starting process" can only be fired if Telegrip is active and the whole system is not operating (R_Telesys.NOT_OP and L_Telesys.NOT_OP). It will later be seen that whenever R_Telesys is in the NOT_OP state, then L_Telesys is in this same state. The specification of the action that reacts to this event is:

```
action start by tt: Telegrip; rts: R_Telesys is
when tt.active and rts.NOT_OP
do
   → rts.STARTING;
   rts.session := rts.session + 1;
end;
```

Event 1.4 Stop Request If the system is supposedly operating or starting (as seen by an observer situated at the Remote location), then it can start the "stopping process". If not, then it is already not operating or in the "stopping process":

```
action stop by rts: R_Telesys is
when rts.OPER or rts.STARTING
do
   → rts.STOPPING;
end;
```

Event 2.1 A Robot Controller Signal is Turned On No change has to be done, as these events are treated properly by system Robot_DifSig (The source code for these actions can be found in A.1.3).

Event 2.2 A Robot Controller Signal is Turned Off No change has to be done, as these events are treated properly by system Robot_DifSig (The source code for these actions can be found in A.1.3. See also a discussion on the assumptions made about the environment events in page 27).

Event 2.3 Motion Detection Like in the case of deactivation of the Telegrip LLTI (event 1.1), we need only modify action motion to start the "stopping process" if necessary. The specification is:

```
refined motion of set_camera_idle by ... lts: L_Telesys is
when ... cm.detecting
```

```
do
   ...
   → lts.camera_idle;
   if (lts.OPER or lts.STARTING) then
      → lts.STOPPING;
   end if;
end;
```

Event 2.4 Robot movement This event is handled properly by the Robot system (see A.1.3). As the Robot object has been placed into the Local location, there is no need to change anything here.

5.3.4 Defining the Partitioned Processes

While developing system Telesys (section 3.4), we grouped the different actions into three groups: starting, operating and stopping actions. In our partitioned system TeleSysLoc, these actions have to be split respecting our choice of locations (section 5.1) and the partitioning of the Telesys class into a local copy, L_Telesys, and a remote one, R_Telesys (section 5.3.1).

As before, we can collect all the resulting split actions into groups which we have called "processes" as there is a notion of several actions cooperating with a common objective. We then have:

- Starting Process: Consists of all actions that cooperate to put TeleSysLoc into an operative state (i.e., R_Telesys.OPER and L_Telesys.OPER). This process is started when action start puts R_Telesys in state STARTING. As the buffer of positions is now split in the two different locations, it is needed to clean both of them before new data can be sent to the robot.

- Operating Process: Consists of the transmission of robot positions from Telegrip to the Robot, including the reading of the data into R_Telesys and its transmission to L_Telesys. As only partial information concerning the state of the whole system is available at either location, the actions in this process will be kept enabled as long as the local information indicates the possibility that the global system is in its operative state. (More on this subject later).

- Stopping Process: Consists of all the actions that cooperate to put TeleSysLoc into an inoperative state (i.e., R_Telesys.NOT_OP and L_Telesys.NOT_OP). The actions of this process stops the other two processes if any is underway.

5.3.5 Specifying the Processes

Each copy of Telesys (R_Telesys and L_Telesys) has states that reflect which process is underway in each location. These states define which actions are enabled for execution in the corresponding location.

The Starting Process

Each time the remote operator invokes the starting procedure, a new session opening is attempted. Each session is assigned a distinctive number, which is communicated to the Local location and is used to distinguish new and valid data from data corresponding to previous sessions. To register the id of the latest session opening attempted, we extend classes R_Telesys and L_Telesys with

$$session : integer := 0;$$

Then "starting process" can be divided into the following steps:

1. If R_Telesys.STARTING, the intention to start the system is communicated to location Local. This communicates the session number as well (com_start).
2. If the Camera is idle, it is started (start_camera).
3. If the Command_Hold signal is on, turn it off (release_hold).
4. Confirm that both the robot and the Camera are ready (ready). This step is needed as a checkpoint for the two previous one, as they can be executed in different orders.
5. Once L_Telesys is ready to start, the current robot position is sent to the Remote location (send_pos). Additionally, the Remote buffer is cleared and a special token is queued so as to tell L_Telesys to discard all previous positions. This token carries the information of the session number whose opening is being attempted.
6. Telesys waits for the special token (in fact we should be considering a union type for position data and this separator. Instead, we have chosen to represent robot positions with non-negative values and reserved negative ones for the transmission of session numbers) which indicates that valid new positions come only after it (wait_confirm). This step involves discarding all positions that arrive from the Remote location before the separator, i.e., it eliminates data corresponding to previous sessions.
7. Upon arrival of the separator, L_Telesys assumes the whole system is operative (start_lts).

The Operating Process

The "operating process" can be divided into the following steps, all cooperating concurrently:

1. R_Telesys acquires a new position from Telegrip, if this has changed (get_new_pos).
2. The head of the Remote buffer is appended to the Local buffer (tx_pkt).
3. The reception of a new position is acknowledged by L_Telesys, which permits the elimination of the first element in the Remote buffer (ack).
4. The position at the head of the Local buffer (if it is not the separator) is used to instruct the robot movement (operate).

Observation 25 *Because we want all actions in TeleSysLoc to modify the contents of variables of only one location at a time,there is no way to remember that a robot position has been copied to the Local buffer. Then, the robot position has to be resent all the time till an action originated in the Local location (ack) tells the sender to stop sending. This attained in the specification with the use of the Alternating Bit Protocol. Therefore, both the R_Telesys and L_Telesys classes were extended with*

$$abp : boolean := false;$$

After the observation, the specification of the actions responsible for the transmission of the robot positions from the Remote location to the Local one, is:

```
action tx_pkt by rts: R_Telesys; lts: L_Telesys is
when rts.buf /= <> and lts.abp = rts.abp and
not(rts.NOT_OP or rts.STOPPING or lts.NOT_OP or lts.STOPPING)
do
    lts.buf := lts.buf & <head( rts.buf )>;
    lts.abp := not(lts.abp);
end;

action ack by rts: R_Telesys; lts: L_Telesys is
```

```
when rts.buf /= <> and lts.abp /= rts.abp
do
    rts.buf := tail(rts.buf);
    rts.abp := not(rts.abp);
end;
```

Observation 26 *As action* **inform_not_op**, *belonging to the "stopping process", changes the contents of the Remote buffer, it has to guarantee that the bits L_Telesys.abp and R_Telesys.abp are equal after its execution.*

The Stopping Process

The "stopping process" is initiated each time R_Telesys or L_Telesys enter their STOPPING states. This transition can be motivated by the execution of any of the following actions: *ack_signal*, *stop*, *motion* or *deact_Telegrip*.

After its initialization, the "starting process" tries to complete the following steps:

1. If the cause of the stoppage is an order from the remote operator (i.e., R_Telesys.STOPPING), then R_Telesys informs L_Telesys the order to stop (comm_stop).
2. The robot is stopped, if it was still running, by setting the signal Command_Hold (stop_r).
3. Once the robot has been stopped, L_Telesys enters its NOT_OP state (rest).
4. L_Telesys informs R_Telesys that it is no longer operating (inform_not_op). As a result, R_Telesys, enters state NOT_OP, as well.

Observation 27 *As action* rest *leaves L_Telesys in state NOT_OP, the system can reach a state Q verifying*

$$R\_Telesys.STARTING \wedge L\_Telesys.NOT\_OP$$

while executing the "stopping process". But Q is also the state of the system after the execution of action start, *so we need some criteria to decide which process to follow from state Q. Our criteria is based on the following appreciation:*

If the system reaches state Q after the execution of action rest, *then* inform_not_op *must be executed (thus completing the "stopping process") before action* com_start *is enabled again.*

L_Telesys and R_Telesys will be extended with a counter. Each time action rest *is executed, this counter is incremented by one in L_Telesys (because* rest *modifies location Local). And each time action* inform_not_op *is executed, the value of the counter in R_Telesys is updated with the value of the counter in L_Telesys. Therefore, action* com_start *is only enabled if both counters agree (while action* inform_not_op *is only enabled if they disagree). Notice finally that these counters can be 2-valued, as the difference between them can be at most one.*

Both classes, R_Telesys and L_Telesys are extended with their respective counters:

$$seq : boolean := false;$$

Then the specification of actions *rest* and *inform_not_op* is:

```
action rest by lts: L_Telesys; r: Robot is
when lts.STOPPING and r.still
do
    → lts.NOT_OP;
    lts.seq := not(lts.seq);
end;
```

action inform_not_op **by** rts: R_Telesys; lts: L_Telesys **is**
when lts.NOT_OP and
(rts.OPER or rts.STOPPING or
(rts.STARTING and lts.seq /= rts.seq))
do
→ rts.NOT_OP;
rts.seq := lts.seq;
rts.abp := lts.abp;
rts.buf := <>;
end;

And the guard of action com_start is strengthened with the conjunct:

$$rts.seq = lts.seq$$

Please, refer to section A.2 for a complete listing of system TeleSysLoc.

5.3.6 Initial Conditions for TeleSysLoc

As with system Telesys (section 3.5), the initial conditions for system TeleSysLoc can be deduced from the initial conditions for each of its classes. We can summarize them with this global DisCo assertion:

initially InitTeleSysLoc **is**
Robot.still.idle
and not(emergency_stop or external_hold
or command_hold or alarm)
and Camera.idle
and Telegrip.inactive
and R_Telesys.not_op and R_Telesys.buf = <> and
R_Telesys.seq = false and R_Telesys.abp = false and
R_Telesys.session = 0
and L_Telesys.not_op and L_Telesys.camera_idle and
R_Telesys.seq = false and R_Telesys.abp = false and
R_Telesys.session = 0 and L_Telesys.buf = <>;

Chapter 6

Property Preservation

In this chapter we apply the ideas of "Synchronized Combination" to establish a relationship between systems *Telesys* and *TeleSysLoc*. This relationship is constructed step by step and a somewhat formal proof is offered showing that it satisfies the requirements of any "synchronized combination" relationship. Finally, we provide a version of the Security Property for system *TeleSysLoc* and use the synchronization relationship to reduce its proof to the almost trivial proof of the *Security Property* for system *Telesys* (Section 3.8).

6.1 A Synchronized Combination of Telesys and TeleSysLoc

A synchronized combination of two systems is essentially a relationship between the states of each system so that we can always pretend that, given a behavior of the more concrete system (*TeleSysLoc* in this case), there exists a behavior of the more abstract system such that the corresponding states of each behavior satisfy the stated relationship.

6.1.1 Objectives

We provide here a synchronized combination of the two systems developed so far, in order to be able to prove an analogous of the *Security Property* for system *TeleSysLoc*. The construction of the synchronization and it's subsequent proof is guided by this objective. As a subsidiary objective, we expect this to be an illustrative example of the use of this methodology.

6.1.2 General Description

In order to define the synchronized combination of *Telesys* and *TeleSysLoc*, each action in *TeleSysLoc* is combined with a number of actions of *Telesys* (possible none). We expect each transition step performed by the more concrete system (*TeleSysLoc*) to be reflected by a legal step of the more abstract system, maybe a stuttering step.

Let us suppose then that we have a "combined" system whose state is composed of a replication of each of the states of *TeleSysLoc* and *Telesys*, where enough provision has been made so as to differentiate variables from both systems with the same name.

Not only do we want that any transition of *TeleSysLoc* be followed by a step of *Telesys*, but we also want no action from *TeleSysLoc* to be restrained by this requirement. That is, we have to prove that, in our combined system, the guards added to the actions derived from *TeleSysLoc* always hold whenever the original action is enabled. This can be reduced to proving that the new guards are implied by the original guards.

In synchronizing the actions of *TeleSysLoc* with those of *Telesys*, we proceed in the opposite direction: For each action of *Telesys*, we choose an action from *TeleSysLoc* to synchronized it

Variable Name	Class
RTS	TeleSysLoc.R_Telesys
LTS	TeleSysLoc.L_Telesys
TS	TeleSys.Telesys
R	TeleSysLoc.Robot
sR	TeleSys.Robot
CM	TeleSysLoc.Camera
sCM	TeleSys.Camera
TT	TeleSysLoc.Telegrip
sTT	TeleSys.Telegrip

Table 6.1: Variables used in the synchronization

with. This could be related to the choosing of a commit point for each "abstract" transaction as in [PD96]. The difference here is that we will not require this commit point to correspond to the action that first changes a specification variable in the transaction (in fact, we don't differentiate specification from implementation variables in this method).

The starting actions of system *Telesys* (*start* and *start_with_cam*) require a special treatment worth mentioning: Their counterpart in *TeleSysLoc* is what we have called "the starting process" (see page 49). The main characteristic of this process is that it can be interrupted by any action leading to the stopping of the whole system ("stopping process", page 51).

6.1.3 Notation

In the following we use TLA, instead of DisCo, to express the actions resulting from the combination of *Telesys* and *TeleSysLoc*. In doing so, we required the DisCo tool to produce a listing with a complete description for each action which was then manually translated into TLA. As both systems employ only one instance of each object, there was no need to introduce quantification over the elements of each class, but we have kept it as a way of indicating which variables are accessed by each action. Furthermore, while doing the combination, we were forced to differentiate variables with the same name but belonging to different systems. In those cases of name clash, we adopted the following convention: Variables belonging to system *TeleSysLoc* retained their original name, while those belonging to *Telesys* were renamed with an 's' appended at the front.

We have also associated a definite class to each variable name, so as to simplify the TLA descriptions. The variables used together with their corresponding class is listed in table 6.1.

6.1.4 Synchronizing Signal Managing Actions

There are two main groups of actions to control and modify the state of robot controller signals: actions that set a given signal and actions that clear it. We consider here only signals "**alarm**", "**external_hold**" and "**emergency_stop**", as the fourth one ("command_hold") is managed by actions that involve other participants besides the robot.

Each *Telesys* action that sets a signal is synchronized with the corresponding action in *TeleSysLoc*. In that way, action *Telesys.set_alarm* is synchronized with *TeleSysLoc.set_alarm*, yielding the following TLA expression:

$$
\begin{array}{l}
Set\_Alarm \doteq \\
\exists R, sR : \\
\quad sR.moving \\
\quad \wedge R.moving \\
\quad \wedge sR.still.signaled' \\
\quad \wedge sR.alarm' \\
\quad \wedge R.still.signaled' \\
\quad \wedge R.alarm' \\
\quad \wedge Unchanged_{Set\_Alarm}
\end{array}
$$

The synchronization of the actions responsible for the clearing of controller signals is not that direct because we have used the occasion to fix the inconvenience caused by the refinement mechanism in DisCo which forced us to define two different actions to clear each signal. For example, in the case of the "**alarm**" signal, Telesys includes two actions to clear it: *clear_alarm1* and *clear_alarm2*. In TLA, we can combine first this two actions into one (similarly with actions in *TeleSysLoc*) and produce the following synchronization:

$$
\begin{array}{l}
Clear\_Alarm \doteq \\
\exists sR, R : \\
\quad sR.alarm \\
\quad R.alarm \\
\quad \wedge \neg sR.Alarm' \\
\quad \wedge if\ \neg(sR.emergency\_stop \vee sR.external\_hold \\
\qquad \vee sR.command\_hold)\ then \\
\qquad sR.still.idle' \\
\quad endif \\
\quad \wedge \neg R.alarm' \\
\quad \wedge if\ \neg(R.emergency\_stop \vee R.external\_hold \\
\qquad \vee R.command\_hold)\ then \\
\qquad R.still.idle' \\
\quad endif \\
\quad \wedge Unchanged_{Clear\_Alarm}
\end{array}
$$

Similar synchronizations were performed with actions that set/clear signals "**external_hold**" and "**emergency_stop**". (See B.3, B.4, B.5, B.6).

6.1.5 Synchronizing the Robot Movement

The robot movement is performed both in system *Telesys* and *TeleSysLoc* by action *move_robot*. As we want both systems to perform the same robot movements, it's just natural to force this two actions to be executed simultaneously. We therefore have:

$$
\begin{array}{l}
Move\_Robot \doteq \\
\exists sR, R : \\
\quad sR.moving \\
\quad R.moving \\
\quad \wedge sR.position' = sR.moving.destination \\
\quad \wedge sR.still.idle' \\
\quad \wedge R.position' = R.moving.destination \\
\quad \wedge R.still.idle' \\
\quad \wedge Unchanged_{Move\_Robot}
\end{array}
$$

Note that we are aiming here at being able to prove that the property

$$\Box(R.position = sR.position) \tag{6.1}$$

holds for the resulting synchronized system. This will allow us to relate the properties of *TeleSysLoc*, related to the robot position, to analogous properties of *Telesys*.

6.1.6 Synchronizing the Operating Actions

In Section 3.4, the main actions of system *Telesys* were collected into three groups: **Starting**, **Stopping** and **Operating** actions. The actions belonging to this last group, *get_new_pos* and *operate*, are treated in a way similar to the synchronization of action *move_robot* in the previous section. That is, the *Telesys* version of each of those actions is fused with the action with the same name from *TeleSysLoc*. The case of *move_model* is slightly different because the guard reflects our assumptions about the environment (Section 3.6.3). In fact, the assumptions regarding the enabledness of action *move_model* are different in *Telesys* than in *TeleSysLoc*. But, as the underlying idea is that conditions under which *TeleSysLoc.move_model* is enabled are more general than those for *Telesys.move_model*, we cannot use the same kind of combination as above. Instead, action *TeleSysLoc.move_model* is synchronized with a stuttering step of *Telesys* if *Telesys.move_model* is not enabled.

Finally, we want property 6.1 to hold and for that we require the model robot to follow the same path in the concrete *Telegrip* (TT) as in the abstract one (sTT), whenever possible.

For the complete description of the resulting synchronized actions, check B.9, B.10 and B.19.

6.1.7 Synchronizing the Starting Actions

Until now, we have been able to synchronize each action of Telesys with its corresponding action (i.e., the action with the same name) from system TeleSysLoc. However, the case of the **Starting** actions of system *Telesys* (*start* and *start_with_cam*) is different. These are actions whose effect in *TeleSysLoc* is implemented not by a single atomic action, but by a so-called "process" (section 5.3.4) consisting of several cooperating actions. Therefore, in order to define the synchronization, we have to identify at least one action of the **Starting** process to match with action *start* (similarly *start_with_cam*).

In choosing one of the actions of the **Starting** process to synchronize with *Telesys.start*, we could have certainly followed [PD96]. In that case, we would have to choose a commit point for the Starting process; in other words, we would have to find the action of *TeleSysLoc* that first modifies variables which also appear in system *Telesys* and then we would have to guarantee that the **Starting** process is carried to its completion.

This suggestion, however, would have prevented the possibility of letting the **Stopping** process interrupt the **Starting** process before the latter is completed. As was extensively discussed in chapter 4, we are not willing to accept such a solution.

Instead we would postpone this "commit point" as much as possible. Ideally the commit point would be at action *TeleSysLoc.start_lts* as it is the action that ends the Starting process by setting RTS into the OPERative state. However, if we did so, the synchronization would not verify the following simple and desirable property:

$$\Box(RTS.oper \Rightarrow TT.active.position = sTT.active.position) \tag{6.2}$$

(The previous property will be necessary to prove the **Security Property** for system *TeleSysLoc*).

Then, instead of synchronizing action *Telesys.start* (or *Telesys.start_with_cam*) with *TeleSysLoc.start_lts*, we synchronize it with *TeleSysLoc.send_pos*. As this last action is the one responsible for updating the position of the model in *Telegrip* (*TT.active.position*) by copying the value of the robot real position (*R.position*), 6.2 will be satisfied.

Finally, notice that actions *start* and *start_with_cam* can be joined into a single action just as we have done with the signal-clearing actions.

The resulting synchronization of *Telesys.start*, *Telesys.start_with_cam* and *TeleSysLoc.send_pos* is:

$Send\_Pos \doteq$
$\exists TS, sTT, sR, LTS, RTS, TT, R:$
$sTT.active \wedge TS.inoperative \wedge sR.still$
$\neg(sR.external\_hold \vee sR.emergency\_stop \vee sR.alarm)$
$LTS.starting \wedge LTS.camera\_detecting \wedge R.still.idle \wedge RTS.starting \wedge TT.active$
$\wedge TS.buf' =<>$
$\wedge if\ TS.inoperative.camera\_off\ then$
$sCM.detecting'$
$endif$
$\wedge TS.operative'$
$\wedge TS.operative.last\_pos' = sR.position$
$\wedge sTT.active.position' = sR.position$
$\wedge mpath' =< sR.position >$
$\wedge rpath' =< sR.position >$
$\wedge RTS.buf' =< (-RTS.session) >$
$\wedge RTS.abp' = LTS.abp$
$\wedge RTS.oper'$
$\wedge RTS.oper.last_{p}os' = R.position$
$\wedge TT.active.position' = R.position$
$\wedge impath' =< R.position >$
$\wedge irpath' =< R.position >$
$\wedge Unchanged_{Send\_Pos}$

6.1.8 Synchronizing the Stopping Actions

The **Stopping Process** in system *TeleSysLoc* is non-interruptible, which means that once it has been initiated, it leads the whole system towards its inoperative state (i.e., a state satisfying $RTS.not\_op \wedge LTS.not\_op$) and, during its execution, no action belonging to either the **Operating** or **Starting** processes is enabled.

The previous observation leaves us plenty of freedom to choose the commit point for each of the **Stopping Actions** in *Telesys*, as no matter what the system does, we can be pretty sure that the Stopping process will be completed (given that the necessary action fairness requirements are satisfied).

However, we should notice that the splitting of actions operated in the transformation of system *Telesys* into system *TeleSysLoc* has allowed new behaviors (in a general sense) that were not allowed by the original specification (Telesys). For example, according to TeleSysLoc, the following situation is possible: Suppose that the system is operative and the robot is moving. The remote user presses the Stop Button, an event that is received at the Remote Location by RTS (action *TeleSysLoc.stop*). Before the order to stop the system reaches the Local Location, motion is detected in the workcell, an event which is received by LTS (action *TeleSysLoc.motion*). It is clear then that we cannot synchronize *TeleSysLoc.stop* with *Telesys.stop* and *TeleSysLoc.motion* with *Telesys.motion_detected*, as the two actions in *Telesys* cannot be executed in succession without having another action start the system in between.

One possible solution would be to synchronize *TeleSysLoc.stop* (resp. *motion*) with a stuttering step, if the system is already inoperative, and with *Telesys.stop* (resp. *motion_detected*), if it is operative.

Our experience with this method was that it resulted in a too complex solution which was too difficult to reason about as the number of possibilities of interleaving actions in the Stopping Process is high. Instead, we looked for a single common point to the execution of any instance of the **Stopping Process**. We chose action *TeleSysLoc.inform_not_op*, as it is always executed in any instance. So we synchronized *inform_not_op* with every **Stopping** action in *Telesys* and with a stuttering step as well. This multiple synchronization, on the other hand, was guided by

a set of relations (between Telesys' and TeleSysLoc's state variables) we wanted to hold at the end of the **Stopping Process**.

Then, the idea is to construct the synchronized action like this:

$$
\begin{aligned}
&Inform\_not\_op \doteq \\
&\exists LTS, RTS, TT, CM, TS, sTT, sCM, sR : \\
&\quad TeleSysLoc.Inform\_not\_op \wedge Rel' \wedge \\
&\quad (Telesys.deact\_telegrip^* \vee Telesys.stop^* \vee \\
&\quad \vee Telesys.motion\_detected^* \vee Telesys.ack\_signal^* \vee \\
&\quad \vee Telesys.deactivate\_telegrip^* \vee Telesys.stuttering^*) \\
&\quad \wedge Unchanged_{Inform\_not\_op}
\end{aligned}
\tag{6.3}
$$

where

- the asterisk indicates the necessary substitution of variables {TT / sTT, CM / sCM, R / sR},
- *Telesys.stuttering* the predicate $TS' = TS \wedge TT' = TT \wedge CM' = CM \wedge R' = R$ and
- *Rel* is a predicate expressing the relationship that we want to hold after the execution of synchronized action *inform_not_op*.

$$Rel \doteq TS.inoperative \wedge (TT.active \Leftrightarrow sTT.active) \wedge (CM.idle \Rightarrow sCM.idle) \tag{6.4}$$

Notice that we cannot require $CM.idle \Leftarrow sCM.idle$ as the Starting Process can be stopped after setting $CM.detecting$ but before executing action Telesys.start_with_cam, thus leaving $sCM.idle$.

As *inform_not_op* is the only action to put the synchronized system into a state satisfying

$$RTS.not\_op \wedge LTS.not\_op \tag{6.5}$$

and any other action preserving 6.5 does not invalidate *Rel*, we would be able to conclude that relation *Rel* will hold in every state satisfying 6.5. In other words, the synchronized system will satisfy

$$\Box(RTS.not\_op \wedge LTS.not\_op \Rightarrow Rel) \tag{6.6}$$

By algebraically manipulating the definition of *inform_not_op* resulting from 6.3, we obtained a more "operational" expression which is easier to translate to DisCo (something that is not done in this work) and that, hopefully, will give more insight to the reader about the characteristics of the action. In particular, the resulting expression shows clearly that the conjuncts added to *TeleSysLoc.inform_not_op* does not add restrictions to its enabledness, as the guard of the action

is not modified.

$$
\begin{array}{l}
Inform\_Not\_Op \doteq \\
\exists LTS, RTS, TT, CM, TS, sTT, sCM, sR : \\
\quad LTS.not\_op \\
\quad (RTS.oper \vee RTS.stopping \vee \\
\qquad (RTS.starting \wedge LTS.seq \neq RTS.seq)) \\
\quad \wedge RTS.not\_op' \\
\quad \wedge RTS.seq' = LTS.seq \\
\quad \wedge if\ (TS.operative \wedge sTT.active) \wedge TT.inactive\ then \\
\qquad sTT.inactive' \\
\qquad sCM.idle' \\
\qquad sR.still.signaled' \\
\qquad sR.command\_hold' \\
\qquad TS.inoperative.camera\_off' \\
\quad else\ if\ TS.operative \wedge \\
\qquad ((sTT.inactive \wedge TT.inactive) \vee \\
\qquad (sTT.active \wedge TT.active)) \\
\quad then \\
\qquad sCM.idle' \\
\qquad sR.still.signaled' \\
\qquad sR.command\_hold' \\
\qquad TS.inoperative.camera\_off' \\
\quad else\ if\ sCM.detecting \\
\qquad ((sTT.inactive \wedge TT.inactive) \vee \\
\qquad (sTT.active \wedge TT.active)) \\
\quad then \\
\qquad sCM.idle' \\
\qquad sR.still.signaled' \\
\qquad sR.command\_hold' \\
\qquad TS.inoperative.camera\_off' \\
\quad else\ if\ (TS.operative \wedge \\
\qquad (sR.external\_hold \vee sR.emergency\_stop \vee sR.alarm)) \wedge \\
\qquad ((sTT.inactive \wedge TT.inactive) \vee \\
\qquad (sTT.active \wedge TT.active)) \\
\quad then \\
\qquad TS.inoperative.camera\_on' \\
\quad else\ if\ (TS.inoperative \wedge sTT.active) \wedge \\
\qquad TT.inactive \wedge (\neg(CM.idle \vee sCM.idle)) \\
\quad then \\
\qquad sTT.inactive' \\
\quad endif \\
\quad \wedge Unchanged_{Inform\_Not\_Op}
\end{array}
\tag{6.7}
$$

6.1.9 Synchronizing the Remaining Actions

At this point there remains just one action of system *Telesys* to synchronize: *activate_telegrip*. As actions *Telesys.deactivate_telegrip* and *Telesys.deact_telegrip* have been synchronized with *TeleSysLoc.inform_not_op*, we cannot guarantee that $TT.active \equiv sTT.active$ hods. Instead, we will be able to prove the restricted property:

$$\Box(TT.active \Rightarrow sTT.active) \tag{6.8}$$

That is, whenever **TT** is activated, **sTT** must be activated if it was not active already. This is done by synchronizing *TeleSysLoc.activate_telegrip* with both, *Telesys.activate_telegrip* and a *stuttering step* of system *Telesys*, depending on the previous state of **sTT**.

Synchronized Action	Telesys Actions	TeleSysLoc Actions
set_alarm	set_alarm	set_alarm
clear_alarm	clear_alarm1 clear_alarm2	clear_alarm1 clear_alarm2
set_external_hold	set_external_hold	set_external_hold
clear_external_hold	clear_external_hold1 clear_external_hold2	clear_external_hold1 clear_external_hold2
set_emergency_stop	set_emergency_stop	set_emergency_stop
clear_emergency_stop	clear_emergency_stop1 clear_emergency_stop2	clear_emergency_stop1 clear_emergency_stop2
move_robot	move_robot	move_robot
move_model	move_model (stuttering step)	move_model
get_new_pos	get_new_pos	get_new_pos
operate	operate	operate
send_pos	start start_with_cam	send_pos
inform_not_op	stop motion_detected ack_signal deactivate_telegrip deact_telegrip (stuttering step)	inform_not_op
activate_telegrip	activate_telegrip (stuttering step)	activate_telegrip
(all remaining actions)	(stuttering step)	(all remaining actions)

Table 6.2: The Synchronization of Telesys and TeleSysLoc

$$
\begin{array}{l}
Activate\_Telegrip \doteq \\
\exists sTT, TT : \\
\quad TT.inactive \\
\quad \wedge if\ sTT.inactive\ then \\
\quad\quad sTT.active' \\
\quad endif \\
\quad \wedge TT.active' \\
\quad \wedge Unchanged_{Activate\_Telegrip}
\end{array}
$$

As a summary, table 6.2 schematically displays the synchronization between systems *Telesys* and *TeleSysLoc* built in this section. Refer to Appendix B for a complete description of the TLA actions composing the synchronization of systems *Telesys* and *TeleSysLoc*.

6.1.10 Initial States of the Synchronized System

The system state is composed of the variables from both Telesys and TeleSysLoc. The initial states of the synchronized system ($Init_{Synchro}$) then, are each composed of an initial state of Telesys (section 3.5) and an initial state of TeleSysLoc, plus a synchronization requirement: we would like the two robot objects to be placed initially in the same position (i.e., $R.position = sR.position$). Thus, we have:

$$
\begin{aligned}
&Init_{Synchro} = \\
&\quad sR.still.idle \\
&\quad \wedge not(sR.emergency\_stop \vee sR.external\_hold \\
&\qquad \vee sR.command\_hold \vee sR.alarm) \\
&\quad \wedge sCM.idle \wedge sTT.inactive \wedge TS.inoperative.camera\_off \wedge TS.buf = <> \\
&\quad \wedge R.still.idle \\
&\quad \wedge not(R.emergency\_stop \vee R.external\_hold \vee R.command\_hold \vee R.alarm) \\
&\quad \wedge CM.idle \wedge TT.inactive \wedge RTS.not\_op \wedge RTS.buf = <> \wedge \neg RTS.abp \\
&\quad \wedge \neg RTS.seq \wedge RTS.session = 0 \\
&\quad \wedge LTS.not\_op \wedge LTS.camera\_idle \wedge LTS.buf = <> \wedge \neg LTS.abp \\
&\quad \wedge \neg LTS.seq \wedge LTS.session = 0 \\
&\quad \wedge R.position = sR.position
\end{aligned}
$$

6.2 The Proof of the Synchronization

As our intention is to use the synchronized combination of systems *Telesys* and *TeleSysLoc* to prove the Security Property, we don't have to prove that it preserves any of the liveness properties of *Telesys* or *TeleSysLoc*. Instead, we have to verify that for every action in the synchronized system constructed in the previous section, the conjuncts of the guard that were obtained from the actions in Telesys always hold whenever the rest of the guard holds.

6.2.1 The Proof Goals

By browsing through the synchronized actions (Appendix B), the proof of the synchronization is reduced to prove each of the following **Goals**:

1. $\Box(R.alarm \Rightarrow sR.alarm)$ (*clear_alarm*, B.2)

2. $\Box(R.external\_hold \Rightarrow sR.external\_hold)$ (*clear_external_hold*, B.4)

3. $\Box(\neg R.emergency\_stop \Rightarrow \neg sR.emergency\_stop)$(*set_emergency_stop*, B.5)

4. $\Box(R.emergency\_stop \Rightarrow sR.emergency\_stop)$ (*clear_emergency_stop*, B.6)

5. $\Box(R.moving \Rightarrow sR.moving)$ (Derived from actions *set_alarm*, *set_external_hold*, *move_robot*. See respectively B.1, B.3, B.7)

6. From action *get_new_pos* (B.10):

   $$
   \begin{aligned}
   &\Box(RTS.oper.last\_pos \neq TT.active.position \Rightarrow \\
   &\quad \Rightarrow TS.operative.last\_pos \neq sTT.active.position)
   \end{aligned}
   $$

7. From action *send_pos* (B.16):

   $$
   \begin{aligned}
   &\Box(LTS.starting \wedge LTS.camera\_detecting \wedge R.still.idle \wedge RTS.starting \wedge TT.active \Rightarrow \\
   &\quad \Rightarrow sTT.active \wedge TS.inoperative \wedge sR.still \\
   &\qquad \wedge \neg(sR.external\_hold \vee sR.emergency\_stop \vee sR.alarm))
   \end{aligned}
   $$

8. From action *operate* (B.19):

   $$
   \begin{aligned}
   &\Box(LTS.oper \wedge LTS.buf \neq <> \wedge R.still \wedge \\
   &\qquad \wedge \neg(R.emergency\_stop \vee R.external\_hold \vee R.alarm) \Rightarrow \\
   &\quad \Rightarrow TS.operative \wedge sR.still \wedge \\
   &\qquad \wedge \neg(sR.emergency\_stop \vee sR.external\_hold \vee sR.alarm))
   \end{aligned}
   $$

6.2.2 Proving some Goals

We proceed now to prove each of the properties just enunciated. For some of the proofs, we will need some previous lemmas. Sometimes, it will also be necessary to prove stronger properties which will provide us with more insight into the kind of relationship established between Telesys and TeleSysLoc.

Proving Goal 1

Proposition 28 $\Box(R.alarm \Leftrightarrow sR.alarm)$

Proof. The proof is straightforward as

$$Init_{Synchro} \Rightarrow \neg R.alarm \wedge \neg sR.alarm$$

and

$$\begin{array}{rcll} set\_alarm & \Rightarrow & R.alarm \wedge sR.alarm & \text{(by B.1)} \\ clear\_alarm & \Rightarrow & \neg R.alarm \wedge \neg sR.alarm & \text{(by B.2)} \end{array}$$

No other action in the synchronized system modifies either $R.alarm$ or $sR.alarm$.
□

Observation 29 *Goal number 1 follows from proposition 28.*

Proving Goals 2, 3 and 4

Proposition 30 $\Box(R.external\_hold \Leftrightarrow sR.external\_hold)$

Proposition 31 $\Box(R.emergency\_stop \Leftrightarrow sR.emergency\_stop)$

The proof of propositions 30 and 31 are similar to that of proposition 28.

Observation 32 *Goal number 2 follows from proposition 30. Goals 3 and 4 follow from 31.*

Proving Goal 5

Lemma 33 $\Box(LTS.not\_op \Rightarrow R.still)$

Proof. Notice first that $Init_{Synchro} \Rightarrow LTS.not\_op \wedge R.still$.
The only action to imply $R.moving'$ is operate, but $operate \Rightarrow LTS.oper'$. And the only action to imply $LTS.not\_op'$ is $rest$, but $rest \Rightarrow R.still'$.
□

Proposition 34 $\Box(R.moving \Rightarrow sR.moving)$

Proof. As $Init_{Synchro} \Rightarrow R.still.idle$, the proposition holds initially. We then have to show that for every synchronized action A, we have

$$(R.moving \Rightarrow sR.moving) \wedge A \Rightarrow (R.moving' \Rightarrow sR.moving')$$

We consider only those actions that could modify the state machines of R or sR:

- $A = operate$. By B.19, $operate \Rightarrow sR.moving'$.
- $A = set\_alarm$, $A = set\_external\_hold$ or $A = set\_emergency\_stop$. By B.1 (resp. B.3, B.5), $A \Rightarrow R.still.signaled' \Rightarrow \neg R.moving'$.

- $A = clear\_alarm$. By B.2, $clear\_alarm \Rightarrow sR.alarm$. It is a property of Telesys that $sR.alarm \Rightarrow \neg sR.moving$. Then, we are only have to prove that

$$\neg R.moving \wedge clear\_alarm \Rightarrow (R.moving' \Rightarrow sR.moving')$$

  which always holds since $\neg R.moving \wedge clear\_alarm \Rightarrow \neg R.moving'$.

- $A = clear\_external\_hold$ or $A = clear\_emergency\_stop$. The proof is similar to that of $clear\_alarm$, as it is verified that

$$\begin{array}{rcl} A & \Rightarrow & sR.alarm \\ \neg R.moving \wedge A & \Rightarrow & \neg R.moving' \end{array}$$

- $A = move\_robot$. By B.7, $move\_robot \Rightarrow R.still' \Rightarrow \neg R.moving$.

- $A = \inf orm\_not\_op$. By B.25, $\inf orm\_not\_op \Rightarrow LTS.not\_op$. Then, by lemma 33, $\inf orm\_not\_op \Rightarrow R.still$ and, as R does not participate in this action, we conclude that $\inf orm\_not\_op \Rightarrow R.still'$.

□

Observation 35 *Goal number 5 follows from proposition 34.*

6.2.3 The proof of an important property

To be able to prove goal 6, we will need to first prove a property relating the position of the robot in *Telesys* to the position of the robot in *TeleSysLoc*. The property if of the utmost importance here as it will play a central rôle in the verification of the Security Property for system TeleSysLoc (see 6.3), as it states that at every state of a behavior of the synchronized system, the two robot positions coincide. In other words, we will prove the following

Proposition 36 $\Box(R.position = sR.position)$

However, several definitions and lemmas will be presented and proved before we are in a position to demonstrate this property.

Lemma 37 $\Box(RTS.oper \Rightarrow TT.active \wedge TS.operative)$

Proof. $Init_{Synchro} \Rightarrow RTS.not\_op$, then the property holds initially. We have to check the invariant only for three groups of actions:

1. Actions that change the state machine of RTS into the *oper* state,
2. Actions that change the state machine of TT, into the *inactive* state and
3. Actions that change the state machine of TS, into the *inoperative* state.

This means that we don't have to consider in the first group, for example, those actions that are only enabled when RTS is already in its *oper* state. In this case, there is only one action in each group:

1. *send_pos*: By B.16, $send\_pos \Rightarrow TT.active' \wedge TS.operative'$.
2. *deactivate_telegrip*: By B.11, we have two cases:
   (a) $(RTS.oper \vee RTS.starting)$:
   $deactivate\_telegrip \wedge (RTS.oper \vee RTS.starting) \Rightarrow RTS.stopping' \Rightarrow \neg RTS.oper$.

(b) $\neg(RTS.oper \vee RTS.starting)$:
$deactivate\_telegrip \wedge \neg(RTS.oper \vee RTS.starting) \Rightarrow RTS' = RTS.$
Then $RTS' = RTS. \wedge \neg(RTS.oper \vee RTS.starting) \Rightarrow \neg RTS.oper$

As in both cases, we have $\neg RTS.oper$, we conclude that the property is invariant under the execution of action *deactivate_telegrip*.

3. *inform_not_op*: By B.25, $inform\_not\_op \Rightarrow RTS.not\_op' \Rightarrow \neg RTS.oper$.

□

Lemma 38 $\Box(TT.active \Rightarrow sTT.active)$

Proof. As $Init_{Synchro} \Rightarrow TT.inactive$, the property holds initially. We have then to consider only those actions that can change the state machines of TT (into the *active* state) or sTT (into the *inactive* state):

- Actions that can effectively change to $sTT.inactive$:
  - *inform_not_op*: By B.25, this change can only occur when $TT.inactive$ and the state of TT is kept unmodified.
- Actions that can effectively change to $TT.active$:
  - *activate_telegrip*: By B.8, $activate\_telegrip \Rightarrow sTT.active'$.
  - *move_model*: By B.9, $move\_model \Rightarrow sTT.active'$

□

Lemma 39 $\Box(RTS.oper \Rightarrow \begin{array}{l} TT.active.position = sTT.active.position \wedge \\ RTS.oper.last\_pos = TS.operative.last\_pos) \end{array}$

Proof. Firstly, notice that according to Lemma 37 if the state of the synchronized systems satisfies $RTS.oper$ then it makes sense to refer to the value $TT.active.position$, as the state also satisfies $TT.active$. Then, according to Lemma 38 the state satisfies $sTT.active$ too, and it makes sense to evaluate $sTT.active.position$.

Secondly, notice that according to Lemma 37 if the state of the synchronized systems satisfies $RTS.oper$ then it makes sense to refer to the value $TS.operative.last\_pos$, as the state also satisfies $TS.operative$.

The property holds initially as $Init_{Synchro} \Rightarrow RTS.not\_op \Rightarrow \neg RTS.oper$.

We only have to verify the invariance of this property for the following two groups of actions:

1. Actions that change the state of RTS into $RTS.oper$:
   - *send_pos*: As this action belongs to the other group too, it is verified there.
2. Actions that modify $TT.active.position$, $sTT.active.position$, $RTS.oper.last\_pos$ and/or $TS.operative.last\_pos$:
   - *send_pos*: By B.16,

$$send\_pos \Rightarrow \begin{array}{l} TT.active.position' = R.position \wedge \\ \wedge sTT.active.position' = sR.position \end{array}$$

But then by Lemma 36, we have

$$R.position = sR.position$$

Notice here that, though we haven't proved Lemma 36 and that its proof will actually use this lemma, there is no circularity here. We are assuming the validity of Lemma 36 at the state holding *before* the execution of *send_pos*, while here we prove a property for the state holding *after* its execution.

And we conclude that

$$send\_pos \Rightarrow TT.active.position' = sTT.active.position' \tag{6.9}$$

Also by B.16,

$$\begin{aligned} send\_pos \Rightarrow \quad & TS.operative.last\_pos' = sTT.active.position' \wedge \\ & \wedge RTS.oper.last\_pos' = TT.active.position' \end{aligned}$$

And, from 6.9, we conclude that

$$send\_pos \wedge RTS.oper \Rightarrow TS.operative.last\_pos' = RTS.oper.last\_pos'$$

- *get_new_pos*: By B.10,

$$\begin{aligned} get\_new\_pos \Rightarrow \quad & TS.operative.last\_pos' = sTT.active.position \wedge \\ & \wedge RTS.oper.last\_pos' = TT.active.position \end{aligned}$$

If the proposition holds in the state from which action *get_new_pos* is executed, then we know that

$$RTS.oper \Rightarrow sTT.active.position \wedge TT.active.position$$

in that state. Therefore, also in that state:

$$send\_pos \wedge RTS.oper \Rightarrow TS.operative.last\_pos' = RTS.oper.last\_pos'$$

Finally notice that action *send_pos* does not change the values of $sTT.active.position$ or $TT.active.position$.

- *move_model*: From the guard of action *move_model* (see B.9), we know that

$$\begin{aligned} move\_model \wedge RTS.oper \Rightarrow \quad & TT.active \wedge \\ & \wedge TT.active.position = RTS.oper.last\_pos \end{aligned}$$

If we assume the proposed property to hold for the state prior to the execution of action *move_model*, then by replacing $sTT.active.position$ for $TT.active.position$ and $TS.operative.last\_pos$ for $RTS.oper.last\_pos$, we have

$$move\_model \wedge RTS.oper \Rightarrow sTT.active.position = TS.operative.last\_pos \tag{6.10}$$

Also, by Lemma 38,

$$move\_model \wedge RTS.oper \Rightarrow sTT.active \tag{6.11}$$

Finally, by B.9, 6.10 and 6.11, the guard of the if-statement is enabled and we then know that

$$move\_model \wedge RTS.oper \Rightarrow TT.active.position' = sTT.active.position'$$

Note also that action *move_model* does not change the values of $RTS.oper.last\_pos$ or $TS.operative.last\_pos$.

- *activate_telegrip*: By B.8 this action can change the values of $TT.active.position$ or $sTT.active.position$, but as it is only enabled when $TT.inactive$, we know that $activate\_telegrip \Rightarrow \neg RTS.oper$ (Lemma 37). This action does not change the state of RTS either.

□

Now we intend to determine a relationship between the values of $TS.buf$ and those of $RTS.buf$ and $LTS.buf$ for some particular states of the synchronized system. In doing so, we will use some special functions whose correct definition will be related to the following lemma:

Lemma 40 $\Box(RTS.abp \not\equiv LTS.abp \Rightarrow RTS.buf \neq <>)$

Proof. This property holds initially because $Init_{Synchro} \Rightarrow \neg RTS.abp \wedge \neg LTS.abp$.

Action *tx_pkt* implies $RTS.buf \neq <>$, and thus, as it does not change it, $RTS.buf' \neq <>$.

And action *ack* implies $RTS.abp \equiv LTS.abp$.

As these are the only actions that can change the state as to validate the antecedent or invalidate the consequence, we are done.

□

Definition 41 *Queue of robot positions in TeleSysLoc. We define a state function that reconstructs the queue of robot positions (plus session separators) in the refined system.*

$$queue(LTS, RTS) = \begin{cases} LTS.buf \ \& \ RTS.buf & if\ (RTS.abp \equiv LTS.abp) \\ LTS.buf \ \& \ tail(RTS.buf) & if\ (RTS.abp \not\equiv LTS.abp) \end{cases}$$

Observation 42 *Notice that Lemma 40 guarantees that state function* **queue** *is well defined.*

Definition 43 *We will say that element x is in the sequence $S =< s_1, s_2, \ldots, s_n >$ and we will write it "x in S" iff there exists i such that $s_i = x$.*

Definition 44 *Function purge eliminates from the sequence S all elements preceding and including the first occurrence of a given element x:*

$$purge(x, S) = purge(x, < s_1, s_2, \ldots, s_n >) ==< s_k, s_{k+1}, \ldots, s_n >$$

where k is the only integer which satisfies

$$(k = n \wedge \neg(x\ in\ S)) \vee (s_{k-1} = x \wedge \neg(x\ in\ < s_1, \ldots, s_{k-2} >)$$

We state without proof some properties of functions **queue** and **purge**:

Lemma 45 $(tx\_pkt \vee ack) \Rightarrow queue(LTS', RTS') = queue(LTS, RTS)$

Lemma 46 $(LTS.buf' = LTS.buf \wedge RTS.abp' = RTS.abp \wedge LTS.abp' = LTS.abp \wedge$
$\wedge RTS.buf' = RTS.buf \ \& < x >) \Rightarrow$
$\Rightarrow queue(LTS', RTS') = queue(LTS, RTS) \ \& < x >$

Lemma 47 $x \neq y \wedge x\ in\ S \Rightarrow purge(x, S \ \& < y >) = purge(x, S) \ \& < y >$

Lemma 48 $x \neq y \wedge x\ in\ S \Rightarrow purge(x, < y > \ \& S) = purge(x, S)$

For the following lemma we will just give a sketch of the proof as it is rather lengthy and analogous to the proof of Lemma 50.

Lemma 49 $\Box \left(\begin{array}{c} (LTS.starting \wedge (-LTS.session)\ in\ queue(LTS, RTS)) \Rightarrow \\ \Rightarrow TS.buf = purge(-LTS.session, queue(LTS, RTS)) \end{array} \right)$

Proof. (Just a sketch)

$Init_{Synchro}$ implies the initial validity of this property.

Its invariance under the execution of actions *tx_pkt* and *ack* is based on Lemma 45 (see the proof of Lemma 50).

For action *send_pos* the proof is easy as this action sets $TS.buf' =<>$, $RTS.abp' = LTS.abp'$ and $RTS.buf' =< -RTS.session >$ while it is not difficult to prove that

$$\Box(LTS.starting \Rightarrow RTS.session = LTS.session)$$

For action *get_new_pos*, the proof follows the corresponding to the same action in proof of Lemma 50, but there we need Lemma 47 instead.

For action *wait_confirm*, follows easily from Lemma 48.

Finally, for action *com_start* (the only action that can change LTS into state *starting*), we have just to prove that

$$\Box(com\_start \Rightarrow (\forall x : x\, in\, queue(LTS', RTS') : x > -LTS.session'))$$

□

Lemma 50 $\Box\,(LTS.oper \Rightarrow TS.buf = queue(LTS, RTS))$

Proof. We only need to show the invariance of this property under the execution of those actions that either leave LTS in state *oper*, or change any of $TS.buf$, $LTS.buf$, $RTS.buf$, $LTS.abp$ or $RTS.abp$. For each of the actions we assume that $LTS.oper \Rightarrow TS.buf = queue(LTS, RTS)$ holds and prove that it implies $LTS.oper' \Rightarrow TS.buf' = queue(LTS', RTS')$.

- *start_lts*: By B.18,

$$\begin{aligned} start\_lts \Rightarrow \quad & LTS.starting \wedge head(LTS.buf) = -LTS.session \wedge \\ & \wedge LTS.buf' = tail(LTS.buf) \end{aligned}$$

  Then, by Lemma 49, $TS.buf = purge(-LTS.session, queue(LTS, RTS))$ and since $head(LTS.buf) = -LTS.session$,

$$TS.buf = \begin{cases} tail(LTS.buf)\ \&\ RTS.buf & if\ (RTS.abp \equiv LTS.abp) \\ tail(LTS.buf)\ \&\ tail(RTS.buf) & if\ (RTS.abp \not\equiv LTS.abp) \end{cases}$$

  But, as $LTS.buf' = tail(LTS.buf)$, finally $TS.buf' = TS.buf = queue(LTS', RTS')$.

- *get_new_pos*: By B.10, we first notice that we have only to consider the case when $LTS.oper$, since this action cannot change it.

  Then, also by B.10

$$\begin{aligned} LTS.buf' = LTS.buf \wedge RTS.abp' = RTS.abp \wedge LTS.abp' = LTS.abp \wedge \\ \wedge RTS.buf' = RTS.buf\ \&\ < TT.active.position > \end{aligned}$$

  and

$$TS.buf' = TS.buf\ \&\ < sTT.active.position >$$

  By Lemma 46 we could conclude that

$$TS.buf' = queue(LTS', RTS')$$

  only if we knew that $TT.active.position = sTT.active.position$. Since $get\_new\_pos \Rightarrow RTS.oper$, this follows from Lemma 39.

- *send_pos*: By B.16, $send\_pos \Rightarrow \neg LTS.oper$.
- *operate*: By B.19, $operate \Rightarrow LTS.oper \wedge LTS.oper' \wedge LTS.buf \neq <>$

  Then, by our assumption, $TS.buf = queue(LTS, RTS)$ and we have to prove that $TS.buf' = queue(LTS', RTS')$.

$$\begin{array}{rcll} TS.buf' & = & tail(TS.buf) & (B.19) \\ & = & tail(queue(LTS, RTS)) & (assumption) \\ & = & tail(queue(LTS, RTS')) & (B.19) \\ & = & tail(LTS.buf)\ \&\ & (LTS.buf \neq <>) \\ & & (if\ LTS.abp \equiv RTS.abp')\ then\ RTS.buf' & \\ & & else\ tail(RTS.buf')) & \\ & = & queue(LTS', RTS')) & (B.19) \end{array}$$

- *tx_pkt* and *ack*: These actions cannot change the state machine in LTS and, by lemma 45, they neither change $queue(LTS, RTS)$.
- *wait_confirm*: By B.17, $wait\_confirm \Rightarrow \neg LTS.oper$.

□

We finally have all necessary elements to prove property 36:

Proof. The property holds initially as $Init_{Synchro} \Rightarrow R.position = sR.position$.
The only action that modifies each of the position values is *move_robot*.
Since by B.7, $move\_robot \Rightarrow \begin{array}{l} R.position' = R.moving.destination \wedge \\ \wedge sR.position' = sR.moving.destination \end{array}$
it suffices to prove that

$$\Box(R.moving \wedge sR.moving \Rightarrow R.moving.destination = sR.moving.destination)$$

This last expression holds initially as $Init_{Synchro} \Rightarrow R.still \wedge sR.still$. And the only action that interests is *operate*:
By B.19, $operate \Rightarrow \begin{array}{l} LTS.oper \wedge LTS.buf \neq <> \wedge \\ \wedge R.moving.destination' = head(LTS.buf) \wedge \\ \wedge sR.moving.destination' = head(TS.buf) \end{array}$
In that case, by 50, we know that

$$head(TS.buf) = head(queue(LTS, RTS) = head(LTS.buf)$$

which proves the property.
□

6.2.4 Proving the Remaining Goals

Proving Goal 6

Proposition 51 $\Box(RTS.oper.last\_pos \neq TT.active.position \Rightarrow \Rightarrow TS.operative.last\_pos \neq sTT.active.position)$

Proof. For the antecedent of the property to be evaluable, we have to assume $RTS.oper$ (Actually this is the assumption in the DisCo tool). In that case, the property follows directly from Lemma 39
□

Observation 52 *Goal number 6 follows from proposition 51.*

Proving Goal 7

Lemma 53 $\Box(RTS.not\_op \vee RTS.starting \Rightarrow TS.inoperative)$

Proof. As $Init_{Synchro} \Rightarrow TS.inoperative$, the property holds initially.

1. Actions that can change TS into state *operative*:

   - *send_pos*: By B.16, $send\_pos \Rightarrow RTS.oper'$

2. Actions that can change RTS into state *not_op* or *starting*:

   - *inform_not_op*: We constructed this synchronized action as to satisfy $inform\_not\_op \Rightarrow Rel'$, where Rel is the relation defined in 6.4 and this implies $TS.inoperative$.
   - *start*: What has been done up to now in this proof suffices to state that $\Box(RTS.not\_op \Rightarrow TS.inoperative)$, and since action *start* is only enabled when $RTS.not\_op$ and it does not modify TS, the property follows.

□

Lemma 54 $\Box((LTS.starting \vee LTS.oper) \wedge R.still \Rightarrow sR.still)$

Proof. As $Init_{Synchro} \Rightarrow LTS.not\_op$, the property holds initially.

1. Actions that can change sR into state *moving*:

   - *operate*: By B.19, $operate \Rightarrow R.moving'$.

2. Actions that can change R into state *still*:

   - *set_alarm, set_emergency_stop, set_external_hold, move_robot*: All these actions imply $sR.still'$.
   - *stop_r*: By B.28, $stop\_r \Rightarrow LTS.stopping'$.

3. Actions that can change LTS into state *starting* or *oper*:

   - *com_start*: By B.13, $com\_start \Rightarrow RTS.starting'$. Then, by Lemma 53, $com\_start \Rightarrow TS.inoperative'$. Finally, it is a property of system *Telesys* that $TS.inoperative \Rightarrow sR.still$.
   - *start_lts*: What has been done up to now in this proof suffices to state that $\Box(LTS.starting \wedge R.still \Rightarrow sR.still)$. Then, by B.18, $start\_lts \Rightarrow LTS.starting$; and, since it does not modify sR, we conclude that $sR.still'$ holds after the execution of this action.

□

Proposition 55
$$\Box(LTS.starting \wedge LTS.camera\_detecting \wedge R.still.idle \wedge RTS.starting \wedge TT.active \Rightarrow \Rightarrow sTT.active \wedge TS.inoperative \wedge sR.still \wedge \neg(sR.external\_hold \vee sR.emergency\_stop \vee sR.alarm))$$

Proof. Because $R.still.idle$ we know that

$$\neg(R.command\_hold \vee R.external\_hold \vee R.emergency\_stop \vee R.alarm)$$

Then, by Propositions 28, 30 and 31, we conclude that

$$\neg(sR.external\_hold \vee sR.emergency\_stop \vee sR.alarm))$$

The rest follows from Lemmas 53, 54 and 38.
□

Observation 56 *Goal number 7 follows from proposition 55.*

Proving Goal 8

Lemma 57 $\Box(LTS.starting \wedge (-LTS.session)\ in\ queue(LTS, RTS) \Rightarrow TS.operative)$

Proof. (Just a sketch) According to Lemma 45, this property remains invariant under actions *tx_pkt* and *ack*. Action *send_pos* is the only one that can add $(-LTS.session)$ to $queue(LTS, RTS)$. It then sets $TS.operative'$. And the only action that changes the TS into state *inoperative* is *inform_not_op*, which is only enabled when $LTS.not\_op$.
□

Lemma 58 $\Box(LTS.oper \Rightarrow TS.operative)$

Proof. Since $Init_{Synchro} \Rightarrow LTS.not\_op$, the property holds initially.

1. Actions that change TS into state *inoperative*:

   - *inform_not_op*: By B.25, $\inf orm\_not\_op \Rightarrow LTS.not\_op'$.

2. Actions that change LTS into state *oper*:

   - *start_lts*: By B.18, $start\_lts \Rightarrow LTS.starting \wedge head(LTS.buf) = -LTS.session$. Therefore, before the execution of this action $(-LTS.session)\ in\ queue(LTS, RTS)$. By Lemma 57 then, we conclude $TS.operative$.

□

Proposition 59
$$\Box(LTS.oper \wedge LTS.buf \neq <> \wedge R.still \wedge \wedge\neg(R.emergency\_stop \vee R.external\_hold \vee R.alarm) \Rightarrow \Rightarrow TS.operative \wedge sR.still \wedge \wedge\neg(sR.emergency\_stop \vee sR.external\_hold \vee sR.alarm))$$

Proof. This follows directly from Lemmas 58, 54, 53, 54 and 38.
□

Observation 60 *Goal number 8 follows from proposition 59.*

By having proved all Proof Goals, we have verified that the system proposed at the beginning of this chapter constitutes in fact a **Synchronized Combination** of systems *Telesys* and *TeleSysLoc*.

6.3 TeleSysLoc Satisfies the Security Property

In order to formally express the Security Property, now of system TeleSysLoc, we follow the same ideas of section 3.8. System TeleSysLoc is thus enriched with two new variables:

- *impath*: The path taken by the robot model as it has been received by R_Telesys (by means of action *get_new_pos*) since the last step verifying

$$\neg R\_Telesys.oper \wedge R\_Telesys.oper'$$

  (this step only occurs when action *send_pos* is executed).

- *irpath*: The path taken by the real robot since the last step verifying the same condition as above.

It is assumed that both variables are initially equal to the empty sequence.

Since the robot can only be moved when $R\_Telesys$ is in its *oper* state, we are only interested in verifying the Security Properties for those states where $R\_Telesys.oper$ holds. We can then express the Security Property for system TeleSysLoc as:

$$\Box(L\_Telesys.oper \Rightarrow impath = irpath \ \&\ queue(L\_Telesys, R\_Telesys)) \tag{6.12}$$

Instead of proving this property directly from the definition of system TeleSysLoc, now that we have established a relationship between this system and system Telesys, we intend to take advantage of the fact that we already know the property to be valid for this latter system (see 3.8).

First, notice that we have already included new variables *impath* and *irpath* in the TLA version of the actions in the synchronized system (see appendix B), though they never affected the proofs of the synchronization relationship given above.

Second, notice that we have included as well the variables *mpath* and *rpath* into the synchronized system. These variables were defined in section 3.8 and used there to express the Security Property (3.8) for system *Telesys*.

Now we prove a simple lemma relating variables *mpath* and *rpath* to *impath* and *irpath*, respectively. This proof depends heavily on properties that have already been proved while verifying the synchronization of systems *Telesys* and *TeleSysLoc*. Notice specially the use of Proposition 36 (section 6.2.3) that assures that both, the robot in system *Telesys* and the robot in system *TeleSysLoc*, are always in the same position.

Lemma 61 $\Box(mpath = impath \wedge rpath = irpath)$

Proof. Initially both variables hold the empty sequence. We then consider only the actions that modify any of these variables:

- *get_new_pos*: By Lemma 39 we know that $TT.active.position = sTT.active.position$. Therefore, $get\_new\_pos \Rightarrow mpath' = impath'$. This action does not change *rpath* or *irpath*.

- *send_pos*: By Proposition 36, we know that $R.position = sR.position$. Then the required equalities hold in the state immediately after the execution of this action.

- *operate*: By Lemma 50 we know that $head(LTS.buf) = head(TS.buf)$. Therefore, $operate \Rightarrow rpath' = irpath'$. This action does not change *mpath* or *impath*.

□

Finally, we have

Proposition 62 *System TeleSysLoc verifies the Security Property (6.12)*

Proof. We prove that this property is satisfied by the synchronized system developed at the beginning of this chapter. Since we have proved that it is in fact a **Synchronized Combination** of systems *Telesys* and *TeleSysLoc*, every behavior of *TeleSysLoc* can be extended to a behavior of the synchronized system. Therefore, a property relating values of the state variables belonging to *TeleSysLoc* that holds for every behavior of the synchronized system will also hold for every behavior of *TeleSysLoc*. Therefore we just have to prove:

$$\Box(LTS.oper \Rightarrow impath = irpath \ \& \ queue(LTS, RTS))$$

Then, if we assume $LTS.oper$, by Lemma 58, $LTS.oper \Rightarrow TS.operative$. And by Lemma 50, $queue(LTS, RTS) = TS.buf$. Combining these with the equalities from Lemma 61, we see that it suffices to prove that

$$\Box(TS.operative \Rightarrow mpath = rpath \ \& \ TS.buf)$$

which we know to hold as it is exactly the Security Property for system *Telesys*.
□

Chapter 7

Conclusions and Future work

7.1 Results

This works describes the results we obtained in attempting the specification of Telesys, the robot teleoperation system demanded by the TELESAFE project.

Two specifications, with different levels of abstraction, were produced. The higher-level specification described Telesys –as well as its hardware and software components– as a closed system in a simple way, but rich enough as to allow for the specification and verification of an important security property (see section 3.8).

Afterwards, the higher-level specification (TeleSysLoc) was transformed into a lower-level one by splitting the atomicity of several actions in the framework of the Synchronized Combination Method (4.8). This was done as a first step towards the introduction of implementation-oriented mechanisms related to the transmission of data and control between distant locations.

During the proof of the Synchronized Combination, several errors were detected in the original lower-level DisCo specification (TeleSysLoc) which would have been quite difficult to detect otherwise.

7.2 Implementation

The robot teleoperation system specified here, Telesys, has been implemented with the C programming language and TCP sockets. The resulting software was used for studying different security concerns in the robot teleoperation environment.

This thesis serves also as documentation for that implementation.

7.3 Specification Language

We applied DisCo to a "real world" problem, which was not originally designed to test the language. The fact that the two DisCo specifications of the robot teleoperation system are relatively short and simple (specially the higher-level one) is more a virtue of DisCo than an indicator of the difficulty posed by the problem. In fact, the C implementation of the system is quite complex and took several months to develop.

The restrictions imposed over the refinement of actions by the definition of the language forced us, on some occasions, to write two actions where it would have been perfectly valid to write only one (see page 19).

When starting from the Telesys specification and trying to derive the TeleSysLoc specification, a complete new system had to be written. Version 2.0 of the language [Dis94] provided no mechanism to reuse Telesys. The latest language version [Dis96] does, but it was not used here as there was no simulation tool implemented for it when this thesis was written.

7.4 Splitting of action atomicity

We have argued for the necessity of action atomicity refinement in the transformational approach to the specification of distributed systems and, in trying to apply it to our specifications, we have analyzed several different methods. In particular, the recent "Aggregation of Distributed Transactions" method (see section 4.7) was shown to be insufficient for our purposes and of little general use if it does not provide a way to link properties from the specifications among themselves. Instead, we have applied an already known Synchronized Combination Method (4.8). Moreover, we have paid special attention to the description and analysis of this method; a work which we humbly consider the main contribution of this thesis.

7.5 Future Work

From now on, work can follow in several directions:

- The proof of the Synchronized Combination of *Telesys* and *TeleSysLoc* done in Chapter 6 showed us that for any reasonably sized system, these proofs can be quite time consuming and error-prone, though they do not present major difficulties. Just because of this simplicity, they are amenable for the application of semi-automatic proof methods.

- The specifications of the robot teleoperation system produced so far are still at too high a level of abstraction and there are plenty of implementation mechanisms that could be introduced as a way of deriving a formal design of the system.

  First, the communication model, which corresponds to the asynchronous message passing style, could be completed. In fact, work has already been done towards the introduction of communication channels as a new step in the transformational specification of Telesys. A particular implementation relation between two specifications has been defined for that purpose and a mechanism to prove the serializability of behaviors of the derived specification has been developed.

  And second, we have assumed communications among locations to be error-free in our model. However, this is not very realistic, as even with the use of protocols such as TCP sockets, connections can brake down resulting in the need for an appropriate reset mechanism. These fault tolerant properties could be added to the specification given here but it could be convenient to have them introduced in the specification from the very beginning (Telesys).

- Finally, regarding the DisCo specification language, it would be interesting to analyze how the work done here would benefit from the new constructs of Version 3.0 [Dis96] and to study how they could be extended to avoid the inconvenient duplication of actions (p. 19).

Bibliography

[AL91] M. Abadi and L. Lamport, The Existence of Refinement Mappings. Theoretical Computer Science, 82(2):253-284, 1991.

[ALM94] M. Abadi, L. Lamport and S. Merz, A TLA Solution to the RPC-Memory Specification Problem. World Wide Web page at http://www.research.digital.com/SRC/dagstuhl/dagstuhl.html. August, 1994.

[AS85] B. Alpern and F. B. Schneider, Defining Liveness. Information Processing Letters, 21(4):181-185, 1985.

[BKs88] R. J. R. Back and R. Kurki-Suonio, Distributed cooperation with actions systems. ACM Trans. Programming Languages Syst. 10, 4, October 1988, pp. 513-554.

[BL94] M. Broy and L. Lamport. The RPC-Memory specification problem. World Wide Web page at http://www.research.digital.com/SRC/dagstuhl/dagstuhl.html. August, 1994.

[ChM88] K. M. Chandy and J. Misra, Parallel Program Design: A Foundation. Addison-Wesley, 1988.

[Dij76] E. W. Dijkstra, A Discipline of Programming. Prentice-Hall, 1976.

[Dis94] The Disco Language Version 2.0. Software Systems Laboratory, Tampere University of Technology. March 11, 1994.

[Dis96] The Disco Language Version 3.0. Software Systems Laboratory, Tampere University of Technology. 1996.

[DNV90] R. De Nicola and F. W. Vaandrager, Three logics for branching bisimulation (extended abstract). In Proceedings 5^{th} Annual Symposium on Logic in Computer Science, Philadelphia, USA, pages 118-129. IEEE Computer Society Press, 1990.

[Ein49] A. Einstein, Authobiographical Notes. Albert Einstein: Philosopher-Scientist. Library of Living Philosophers. Open Court Publishing Company, La Salle, Illinois, U.S.A., 1949.

[Jar92] H.-M. Järvinen, The design of a specification language for reactive systems. Tampere University of Technology, Publication 95, 1992.

[JKs91] H.-M. Järvinen and R. Kurki-Suonio, DisCo specification language: Marriage of actions and objects. Proc. 11th Conf. on Distributed Computing Syst., 1991, pp. 142-151.

[JKSS90] H.-M. Järvinen, R. Kurki-Suonio, M. Sakkinen and K. Systä, Object-oriented specification of reactive systems. Proc. 12th Int. Conf. on Software Eng., 1990, pp 63-71.

[Kel94] P. Kellomäki, Analysis of a Stabilizing Protocol: A Case Study in Reasoning about Action Systems. Thesis for the Degree of Licentiate of Technology. Software Systems Laboratory, Tampere University of Technology. December, 1994.

[Ks95] R. Kurki-Suonio, Incremental Specification with Joint Actions: The RPC-Memory Specification Problem.Software Systems Laboratory, Tampere University of Technology. 1995

[Ks96] R. Kurki-Suonio, Notes for the course "Specification of Reactive Systems ". Software Systems Laboratory, Tampere University of Technology. January, 1996.

[Ks96b] R. Kurki-Suonio, Fundamentals of Object-Oriented Specification and Modeling of Collective Behaviors. Software Systems Laboratory, Tampere University of Technology. 1996.

[KsK88] R. Kurki-Suonio and T.Kankaanpää, On the Design of Reactive Systems. Software Systems Laboratory, Tampere University of Technology. Report 1. October, 1988.

[LLOR96] P. B. Ladkin, L. Lamport, B. Olivier, and D. Roegel. Lazy caching: An assertional view. Distributed Computing, 1996.

[Lam83] L. Lamport, Specifying concurrent program modules. ACM Transactions on Programming Languages and Systems, 5(2):190-222, 1983.

[Lam88] L. Lamport, A Theorem on Atomicity in Distributed Algorithms. Digital Equipment Corporation. May, 1988.

[Lam94] L. Lamport, The Temporal Logic of Actions. ACM Trans. Prog. Lang. Syst., 16(3):872-923, May, 1994.

[Lam96] L. Lamport, Refinement in State-Based Formalisms. Digital Equipment Corportation. December, 1996.

[LV94] N. Lynch and F. Vaandrager, Forward and Backward Simulations - part I: Untimed Systems. October, 1994.

[Lyn94] N. Lynch, Atomic Transactions for Multiprocessor Programming: A Formal Approach. DIMACS Series in Discrete Math. and Theoretical Comp. Sc., 1994.

[Mik95] T. Mikkonen, Implementation of Reactive Systems based on Closed-system Specifications. Software Systems Laboratory, Tampere University of Technology. Licentiate thesis. June, 1995.

[PaG96] E. Pascal Gribomont, Atomicity Refinement and Trace Reduction Theorems. Computer Aided Verification, 8^{th} International Conference, CAV'96, pp. 311-322, LNCS 1102, Springer-Verlag, July 1996.

[PD96] S. Park and D. L. Dill, Protocol Verification by Aggregation of Distributed Transactions. Computer Aided Verification, 8^{th} International Conference, CAV'96, pp. 299-310, LNCS 1102, Springer-Verlag, July 1996.

[Sys91] K. Systä, A graphical tool for the specification of reactive systems. Proc. Euromicro'91 Workshop on Real-Time Systems, 1991, pp. 12-19.

Appendix A

Telesys Source Code

A.1 The initial system

A.1.1 The Robot system

```
-- DisCo specification of Telesys Robot Teleoperation System.
--
--
--      R O B O T      S Y S T E M
--
--
-- Author: Pablo Giambiagi
-- Date : 27.03.96
--
-- Observations:
--
--      This is a highly simplified specification of a robot.
--
--      At any moment, the robot can be in any of two states: moving or still.
--      If it is moving, then it is moving towards a target position
--      (destination).
--
--      The current position of the robot can be acquired at any time. This
--      value has been simplified to be just an integer.
--
--      The actual path taken by the robot to go from it's current position
--      to its destination is abstracted away.
--
--      The only events considered are:
--      1. Starting off towards a new position (action Set_Destination).
--      2. Reaching destination (action move_Robot).
--      3. Stoppage of the robot by any other circumstance,
--      e.g. "emergency stop" (action stop_Robot). Notice that a robot
--      can be stopped even when it is already still. In this case, there
--      is no associated state-change.

system Robot is

    class Robot is
      state *still, moving( destination: integer );
```

```
        position: integer;
        initially position >= 0;

•       assert position >= 0;
    end;

-- Actions ----------------------------

    action set_Destination( p: integer) by r: Robot is
    when r.still and p >= 0
    do
        → r.moving( p );
    end;

    action stop_Robot by r: Robot is
    when true
    do
        → r.still;
    end;

    action move_Robot by r: Robot is
    when r.moving
    do
        r.position := r.moving.destination;
        → r.still;
    end;

end;
```

A.1.2 The Robot_Signal System

```
-- DisCo specification of Telesys Robot Teleoperation System.
--
--
--          R O B O T _ S I G N A L
--
--
-- Author: Pablo Giambiagi
-- Date : 01.04.96
--
-- Observations:
--
--   1. Robot system extended with robot controller signals.
--   2. Every time a signal is received by the robot controller,
--      the robot is immediately stopped.
--   3. Robot state "still" is extended to distinguish between these
--      situations:
--      a) Robot stopped after completion of correct movement
--         (imported action move_Robot)
--      b) Robot stopped after reception of signal.
--   4. If the robot controller has received a signal, then the signal
--      has to be cleared before the robot can perform any movement.
--      However, this restriction is not introduced at this level, as
```

```
-- the process needed to clear that signal depends on the type of
-- the signal (see system Robot_DifSig).
-- 5. Two new events are considered:
-- a) Turning a signal on (set_signal)
-- b) Clearing all signals (clear_signal).
-- 6. Signals are not distinguished. Anyway, there can be several
-- different types of signals. Accordingly, action set_signal is
-- always enabled.

system Robot_Signal

import Robot;

is

extend Robot by
    extend still by
        state *idle, signaled;
    end still;
end;
-- Actions ----------------

refined move_Robot is
when ...
do
    ...
    → r.still.idle;
end;

action set_signal by r: Robot is
when true
do
    → r.still.signaled;
end;

action clear_signal by r: Robot is
when r.still.signaled
do
→ r.still.idle;
end;

end;
```

A.1.3 The Robot_DifSig System

```
-- DisCo specification of Telesys Robot Teleoperation System.
--
--        R O B O T _ D I F S I G
--
-- Author: Pablo Giambiagi
-- Date : 26.08.96
--
-- Observations:
--
```

```
-- 1. Robot system plus signals, now differentiated.
-- 2. The robot controller recognizes four different signals:
--    a) Emergency Stop
--    b) External Hold
--    c) Command Hold : This signal is raised when the robot is stopped
--    by a software command. As this is the only way available to the
--    Teleoperation System to stop the robot, the guard of the
--    imported action "stop_robot" is strengthened to false.
--    d) Alarm : This signal can result from an error internal to the
--       controller or, for example, the impossibility to reach a certain
--       position.
-- 3. A refinement in Disco forces a complete inclusion of the original
--    body in every refined action. That is, the original body cannot be
--    guarded by an IF command. This has forced us to split every action
--    to clear a signal into two actions, taking into account the state of
--    the other signals.

system Robot_DifSig

   import Robot_Signal;

is

   extend Robot by
      -- The four different signals recognized by the robot controller.
      -- Notice that this could be defined as extensions to the state
      -- still.signaled, but it would have made the notation rather clumsier.
      emergency_stop: boolean;
      external_hold: boolean;
      command_hold: boolean;
      alarm: boolean;

      -- The following asserts that any signal implies the state still.signaled
      assert not(emergency_stop or external_hold or command_hold or alarm) or
         still.signaled;
   end;

-- Actions ----------------------
-- Signal setting actions:

   refined set_emergency_stop of set_signal is
   when ... not(r.emergency_stop)
   do
      ...
      r.emergency_stop := true;
   end;

   refined set_external_hold of set_signal is
   when ... r.moving
   do
   ...
   r.external_hold := true;
   end;
```

```
    -- The guard of this actions does not require "not(r.command_hold)"
    -- because it will later be used in a 'combined' clause and we don't
    -- expect that guard to propagate to that system (Telesys). Without the
    -- guard, the execution of this action results in a stuttering step.
    refined hold_on of set_signal is
    when ...
    do
        ...
        r.command_hold := true;
    end;

    refined set_alarm of set_signal is
    when ... r.moving
    do
        ...
        r.alarm := true;
    end;

-- Signal clearing actions:

    -- Note that DisCo's own restrictions have forced us to implement
    -- each event as a pair of actions. We would have liked to be able
    -- to write, for example:
    --
    -- refined clear_emergency_stop1 of clear_signal is
    -- when ... r.emergency_stop
    -- do
    --     if not(r.external_hold or r.command_hold or r.alarm) then
    --         ...
    --     end if;
    --     r.emergency_stop := false;
    -- end;

    refined clear_emergency_stop1 of clear_signal is
    when ... r.emergency_stop and
            not(r.external_hold or
            r.command_hold or
            r.alarm)
    do
        ...
        r.emergency_stop := false;
    end;

    action clear_emergency_stop2 by r: Robot is
    when r.emergency_stop and
            (r.external_hold or r.command_hold or r.alarm)
    do
        r.emergency_stop := false;
    end;

    refined clear_external_hold1 of clear_signal is
    when ... r.external_hold and
            not(r.emergency_stop or
            r.command_hold or
```

```
        r.alarm)
do
    ...
    r.external_hold := false;
end;

action clear_external_hold2 by r: Robot is
when r.external_hold and
        (r.emergency_stop or r.command_hold or r.alarm)
do
    r.external_hold := false;
end;

refined hold_off1 of clear_signal is
when ... not(r.emergency_stop or
        r.external_hold or
        r.alarm)
do
    ...
    r.command_hold := false;
end;

action hold_off2 by r: Robot is
when r.emergency_stop or r.external_hold or r.alarm
do
    r.command_hold := false;
end;

refined clear_alarm1 of clear_signal is
when ... r.alarm and
        not(r.emergency_stop or
        r.external_hold or
        r.command_hold)
do
    ...
    r.alarm := false;
end;

action clear_alarm2 by r: Robot is
when r.alarm and (r.emergency_stop or r.external_hold or r.command_hold)
do
    r.alarm := false;
end;

-- The action that sets the robot movement is able to clear a command_hold.
refined set_Destination is
when ... not(r.emergency_stop or r.external_hold or r.alarm)
do
    ...
    r.command_hold := false;
end;

-- The original actions are permanently disabled.
refined set_signal is
```

```
        when ... false
        do
            ...
        end;

        refined clear_signal is
        when ... false
        do
            ...
        end;

        -- The functionality of the "stop_robot" action is replaced by
        -- that of action "hold_on".
        refined stop_robot is
        when ... false
        do
            ...
        end;

    end;
```

A.1.4 The Camera System

```
-- DisCo specification of Telesys Robot Teleoperation System.
--
--          C A M E R A
--
-- Author: Pablo Giambiagi
-- Date : 28.03.96
--
-- Observations:
--
--      This systems represents the camera used to detect movements inside
--      the robot workcell.
system Camera

is

    class Camera is
        state *idle, detecting;
    end;

-- Actions ----------------------

    action set_camera_idle by cm: Camera is
    when true
    do
        → cm.idle;
    end;

    action start_detection by cm: Camera is
    when true
    do
        → cm.detecting;
```

```
    end;

end;
```

A.1.5 The Telegrip System

```
-- DisCo specification of Telesys Robot Teleoperation System.
--
--          T E L E G R I P
--
-- Author: Pablo Giambiagi
-- Date : 27.03.96
--
-- Observations:
--
--    1. This system represents the Teleoperation simulation software
--       where the remote operator directs a 3D model of the robot and its
--       workcell.
--    2. From the point of view of the Teleoperation System, it is only of
--       concern to know the state of the LLTI associated to the model (i.e.
--       LLTI active or inactive) and, if the LLTI is active, the position
--       of the robot according to the Telegrip model.
--    3. Being part of the environment of the Teleoperation System, no
--       restrictions are imposed to the action responsible for the movement
--       of the robot model.

system Telegrip

is

    class Telegrip is
        state *inactive, active( position: integer );
    end;
-- Actions ------------------------------

    action activate_Telegrip by tt: Telegrip is
    when tt.inactive
    do
        -> tt.active;
    end;

    action deactivate_Telegrip by tt: Telegrip is
    when tt.active
    do
        -> tt.inactive;
    end;

    action change_Pos( p: integer ) by tt: Telegrip is
    when tt.active and p >= 0
    do
        -> tt.active( p );
    end;
```

```
end;
```

A.1.6 The Telesys System

```
-- DisCo specification of Telesys Robot Teleoperation System.
--
--          T E L E S Y S
--
-- Author: Pablo Giambiagi
-- Date : 19.04.96
-- Modified: 25.04.96
--
-- Initial TeleSys system (with a simple unbounded buffer)

system TeleSys

    import Telegrip, Camera, Robot_DifSig;


combined start_with_cam of Telegrip.change_Pos,
                           Camera.start_detection;

combined stop of Robot_DifSig.hold_on,
                 Camera.set_camera_idle;

combined deact_Telegrip of Robot_DifSig.hold_on,
                           Camera.set_camera_idle,
                           Telegrip.deactivate_Telegrip;

is

    class TeleSys is
        state *inoperative, operative(last_pos: integer);
        extend inoperative by
            state *camera_off, camera_on;
        end inoperative;
        buf: sequence integer;
        initially buf = <>;
    end;

-- Actions ----------------------------------
- Starting...

    refined start of change_Pos by ... ts: TeleSys; r: Robot is
    when ... ts.inoperative.camera_on and
            not(r.external_hold or r.emergency_stop or r.alarm) and
            r.still and p = r.position
    do
        ts.buf := <>;
        → ts.operative(p);
        ...
    end;

    refined start_with_cam by ... ts: TeleSys; r: Robot is
    when ... ts.inoperative.camera_off and
```

```
        not(r.external_hold or r.emergency_stop or r.alarm) and
        r.still and p = r.position
    do
        ts.buf := <>;
        → ts.operative(p);
        ...
    end;

---------
- - Operating...

    refined move_Model of change_Pos by ... ts: Telesys is
    when ... (ts.inoperative or tt.active.position = ts.operative.last_pos)
    do
        ...
    end;

    action get_new_pos by ts: Telesys; tt: Telegrip is
    when ts.operative.last_pos /= tt.active.position
    do
        -¿ ts.operative(tt.active.position);
        ts.buf := ts.buf & <tt.active.position>;
    end;

    refined operate of set_Destination
    by ... ts: TeleSys is
    when ... ts.operative and
            p = head(ts.buf)
    do
        ...
        ts.buf := tail(ts.buf);
    end;

---------
- Stopping...

    action ack_Signal by r: Robot; ts: Telesys is
    when ts.operative and
            (r.external_hold or r.emergency_stop or r.alarm)
    do
        → ts.inoperative.camera_on;
    end;

    refined motion_detected of stop by ... ts: Telesys is
    when ... cm.detecting
    do
        ...
        → ts.inoperative.camera_off;
    end;

    refined stop by ... ts: Telesys is
    when ... ts.operative
    do
        ...
```

```
        → ts.inoperative.camera_off;
    end;

    refined deact_Telegrip by ... ts: Telesys is
    when ... ts.operative
    do
        ...
        → ts.inoperative.camera_off;
    end;

    refined deactivate_Telegrip by ... ts: Telesys is
    when ... ts.inoperative
    do
        ...
    end;

    ---------------------------
    -- Some actions are no longer valid

    refined start_detection is
    when ... false
    do
        ...
    end;

    refined set_camera_idle is
    when ... false
    do
        ...
    end;

    refined hold_on is
    when ... false
    do
        ...
    end;

    refined hold_off1 is
    when ... false
    do
        ...
    end;

    refined hold_off2 is
    when ... false
    do
        ...
    end;

end;

creation C_TELESYS of TeleSys is
    new Robot;
    new Camera;
```

```
    new Telegrip;
    new TeleSys;
end C_TELESYS;
```

A.2 The TelesSysLoc System

```
-- DisCo specification of Telesys Robot Teleoperation System.
--
--    T E L E S Y S L O C
--
-- Author: Pablo Giambiagi
-- Date : 26.08.96
-- Last modified: 28.08.96
--
-- TeleSys system with separate (Local and Remote) locations.

system TeleSysLoc

    import Telegrip, Camera, Robot_DifSig;

is

    class R_TeleSys is
        state *NOT_OP, STARTING, STOPPING, OPER(last_pos: integer);
        buf: sequence integer := <>;
        seq: boolean := false;
        abp: boolean := false;
        session: integer := 0;
    end;

    class L_TeleSys is
        state *NOT_OP, STARTING, STOPPING, OPER;
        state *camera_idle, camera_detecting;
        buf: sequence integer := <>;
        seq: boolean := false;
        abp: boolean := false;
        session: integer := 0;
    end;

-- Actions ----------------------------------
-- Starting the system...

    action start by tt: Telegrip; rts: R_Telesys is
    when tt.active and rts.NOT_OP
    do
        → rts.STARTING;
        rts.session := rts.session + 1;
    end;

    -- Communicating the intention to start the system to
    -- the local Location.
    action com_start by rts: R_Telesys; lts: L_Telesys is
    when rts.STARTING and lts.NOT_OP and rts.seq = lts.seq
```

```
do
    → lts.STARTING(false);
    lts.session := rts.session;
end;

-- Start the camera if it is idle and L_Telesys wants to start.
refined start_camera of start_detection by ... lts: L_Telesys is
when ... lts.camera_idle and lts.STARTING
do
    → lts.camera_detecting;
    ...
end;

-- Release hold if robot has to (and can) be restarted.
refined release_hold of hold_off1 by ... lts: L_Telesys is
when ... lts.STARTING
do
    ...
end;

-- When ready, L_Telesys sends the robot position to update Telegrip.
refined send_pos of change_Pos
by ... rts: R_Telesys; lts: L_Telesys; r: Robot is
when ... lts.STARTING and lts.camera_detecting and
        r.still.idle and rts.STARTING and p = r.position
do
    rts.buf := < (- rts.session) > ; — a token to clear the channel
    → rts.OPER(p);
    rts.abp := lts.abp;
    ...
end;

-- While waiting for the confirmed-start message, all positions
-- received are eliminated.
action wait_confirm by lts: L_Telesys is
when lts.STARTING and head(lts.buf) /= - lts.session
do
    lts.buf := tail(lts.buf);
end;

-- Reception of the confirmed-start message.
action start_lts by lts: L_Telesys is
when lts.STARTING and head(lts.buf) = - lts.session
do
    → lts.OPER;
    lts.buf := tail(lts.buf);
end;
```

```
-- Stopping the system...

-- Recognize a robot exception condition.
action ack_Signal by r: Robot; lts: L_Telesys is
when r.still.signaled and (lts.OPER or lts.STARTING)
```

```
do
   → lts.STOPPING;
end;

-- Stop the robot if it is not already stopped.
refined stop_r of hold_on by ... lts: L_Telesys is
when ... lts.STOPPING
do
   ...
end;

-- When the camera detects movements inside the workcell, it
-- informs (interrupts) immediately L_Telesys.
refined motion of set_camera_idle by ... lts: L_Telesys is
when ... cm.detecting
do
   ...
   → lts.camera_idle;
   if (lts.OPER or lts.STARTING) then
      → lts.STOPPING;
   end if;
end;

-- After L_Telesys has stopped the robot, it can rest.
action rest by lts: L_Telesys; r: Robot is
when lts.STOPPING and r.still
do
   → lts.NOT_OP;
   lts.seq := not(lts.seq);
end;

-- If L_Telesys has not accomplished a complete start, it tells
-- R_Telesys.
action inform_not_op by rts: R_Telesys; lts: L_Telesys is
when lts.NOT_OP and
      (rts.OPER or rts.STOPPING or
      (rts.STARTING and lts.seq /= rts.seq))
do
   → rts.NOT_OP;
   rts.seq := lts.seq;
end;

-- The system can be stopped by user request.
action stop by rts: R_Telesys is
when rts.OPER or rts.STARTING
do
   → rts.STOPPING;
end;

-- Deactivating Telegrip also stops the system.
refined deactivate_Telegrip by ... rts: R_Telesys is
when ...
do
   ...
```

```
    if (rts.OPER or rts.STARTING) then
        → rts.STOPPING;
    end if;
end;

-- Once R_Telesys enters its STOPPING state, it should communicate it
-- to L_Telesys.
action comm_stop by rts: R_Telesys; lts: L_Telesys is
when rts.STOPPING and not(lts.STOPPING or lts.NOT_OP)
do
    → lts.STOPPING;
end;
```

```
- Operating the system...

    -- Because the idea is never to write to more than one location at the
    -- same time, there is no way to remember that a message has been sent.
    -- Then, the message has to be resent all the time till an action
    -- originated in another location (ack) tells the sender to stop
    -- sending that packet.

    action tx_pkt by rts: R_Telesys; lts: L_Telesys is
    when rts.buf /= <> and lts.abp = rts.abp and
            not(rts.NOT_OP or rts.STOPPING or lts.NOT_OP or lts.STOPPING)
    do
        lts.buf := lts.buf & <head(rts.buf)>;
        lts.abp := not(lts.abp);
    end;

    action ack by rts: R_Telesys; lts: L_Telesys is
    when rts.buf /= <> and lts.abp /= rts.abp
    do
        rts.buf := tail(rts.buf);
        rts.abp := not(rts.abp);
    end;

    refined move_Model of change_Pos by ... rts: R_Telesys is
    when ... (not(rts.OPER) or tt.active.position = rts.OPER.last_pos)
    do
        ...
    end;

    action get_new_pos by rts: R_Telesys; tt: Telegrip is
    when rts.OPER.last_pos /= tt.active.position
    do
        → rts.OPER(tt.active.position);
        rts.buf := rts.buf & <tt.active.position>;
    end;

    refined operate of set_Destination
    by ... lts: L_TeleSys is
    when ... lts.OPER and p = head(lts.buf)
    do
```

```
        ...
        lts.buf := tail(lts.buf);
    end;

-------------------------------------------------
-- Some actions are no longer valid

    refined set_camera_idle is
    when ... false
    do
        ...
    end;

    refined hold_on is
    when ... false
    do
    ...
    end;

    refined hold_off2 is
    when ... false
    do
        ...
    end;

end;

creation C_TELESYSLOC of TeleSysLoc is
    new Robot;
    new Camera;
    new Telegrip;
    new R_TeleSys;
    new L_TeleSys;
end C_TELESYSLOC;
```

Appendix B

The Synchronized System in TLA

$$
\begin{array}{l}
Set\_Alarm \doteq \\
\exists sR, R : \\
\quad sR.moving \\
\quad R.moving \\
\quad \wedge sR.still.signaled' \\
\quad \wedge sR.alarm' \\
\quad \wedge R.still.signaled' \\
\quad \wedge R.alarm' \\
\quad \wedge Unchanged_{Set\_Alarm}
\end{array}
\tag{B.1}
$$

$$
\begin{array}{l}
Clear\_Alarm \doteq \\
\exists sR, R : \\
\quad sR.alarm \\
\quad R.alarm \\
\quad \wedge \neg \acute{s}R.Alarm' \\
\quad \wedge if\ \neg(sR.emergency\_stop \vee sR.external\_hold \\
\qquad \vee sR.command\_hold)\ then \\
\qquad sR.still.idle' \\
\quad endif \\
\quad \wedge \neg R.alarm' \\
\quad \wedge if\ \neg(R.emergency\_stop \vee R.external\_hold \\
\qquad \vee R.command\_hold)\ then \\
\qquad R.still.idle' \\
\quad endif \\
\quad \wedge Unchanged_{Clear\_Alarm}
\end{array}
\tag{B.2}
$$

$$
\begin{array}{l}
Set\_External\_Hold \doteq \\
\exists sR, R : \\
\quad sR.moving \\
\quad R.moving \\
\quad \wedge sR.still.signaled' \\
\quad \wedge sR.external\_hold' \\
\quad \wedge R.still.signaled' \\
\quad \wedge R.external\_hold' \\
\quad \wedge Unchanged_{Set\_External\_Hold}
\end{array}
\tag{B.3}
$$

$$
\begin{array}{l}
Clear\_External\_Hold \doteq \\
\exists sR, R : \\
\quad sR.external\_hold \\
\quad R.external\_hold \\
\quad \wedge \neg sR.external\_hold' \\
\quad \wedge if\ \neg(sR.emergency\_stop \vee sR.alarm \\
\qquad \vee sR.command\_hold)\ then \\
\qquad sR.still.idle' \\
\quad endif \\
\quad \wedge \neg R.external\_hold' \\
\quad \wedge if\ \neg(R.emergency\_stop \vee R.alarm \\
\qquad \vee R.command\_hold)\ then \\
\qquad R.still.idle' \\
\quad endif \\
\quad \wedge Unchanged_{Clear\_External\_Hold}
\end{array}
\tag{B.4}
$$

$$
\begin{array}{l}
Set\_Emergency\_Stop \doteq \\
\exists sR, R : \\
\quad \neg(sR.emergency\_stop) \\
\quad \neg(R.emergency\_stop) \\
\quad \wedge sR.still.signaled' \\
\quad \wedge sR.emergency\_stop \\
\quad \wedge R.still.signaled' \\
\quad \wedge R.emergency\_stop \\
\quad \wedge Unchanged_{Set\_Emergency\_Stop}
\end{array}
\tag{B.5}
$$

$$
\begin{array}{l}
Clear\_Emergency\_Stop \doteq \\
\exists sR, R : \\
\quad sR.emergency\_stop \\
\quad R.emergency\_stop \\
\quad \wedge \neg sR.emergency\_stop' \\
\quad \wedge if\ \neg(sR.external\_hold \vee sR.alarm \\
\qquad \vee sR.command\_hold)\ then \\
\qquad sR.still.idle' \\
\quad endif \\
\quad \wedge \neg R.emergency\_stop' \\
\quad \wedge if\ \neg(R.external\_hold \vee R.alarm \\
\qquad \vee R.command\_hold)\ then \\
\qquad R.still.idle' \\
\quad endif \\
\quad \wedge Unchanged_{Clear\_Emergency\_Stop}
\end{array}
\tag{B.6}
$$

$$
\begin{array}{l}
Move\_Robot \doteq \\
\exists sR, R : \\
\quad sR.moving \\
\quad R.moving \\
\quad \wedge sR.position' = sR.moving.destination \\
\quad \wedge sR.still.idle' \\
\quad \wedge R.position' = R.moving.destination \\
\quad \wedge R.still.idle' \\
\quad \wedge Unchanged_{Move\_Robot}
\end{array}
\tag{B.7}
$$

$$
\begin{array}{l}
Activate\_Telegrip \doteq \\
\exists sTT, TT : \\
\quad TT.inactive \\
\quad \wedge if\ sTT.inactive\ then \\
\quad\quad sTT.active' \\
\quad endif \\
\quad \wedge TT.active' \\
\quad \wedge Unchanged_{Activate\_Telegrip}
\end{array}
\tag{B.8}
$$

$$
\begin{array}{l}
Move\_Model \doteq \\
\exists TS, sTT, RTS, TT : \\
\quad \exists P : \\
\quad TT.active \wedge P >= 0 \\
\quad (\neg(RTS.oper) \vee \\
\quad\quad TT.active.position = RTS.oper.last\_pos) \\
\quad \wedge if\ sTT.active \wedge (TS.inoperative \vee \\
\quad\quad sTT.active.position = TS.operative.last\_pos)\ then \\
\quad\quad\quad sTT.active' \\
\quad\quad\quad sTT.active.position' = P \\
\quad endif \\
\quad \wedge TT.active' \\
\quad \wedge TT.active.position' = P \\
\quad \wedge Unchanged_{Move\_Model}
\end{array}
\tag{B.9}
$$

$$
\begin{array}{l}
Get\_New\_Pos \doteq \\
\exists TS, sTT, RTS, TT : \\
\quad TS.operative.last\_pos \neq sTT.active.position \\
\quad RTS.oper.last\_pos \neq TT.active.position \\
\quad \wedge TS.operative.last\_pos' = sTT.active.position \\
\quad \wedge TS.buf' = TS.buf\ \&\ < sTT.active.position > \\
\quad \wedge RTS.oper.last\_pos' = TT.active.position \\
\quad \wedge RTS.buf' = RTS.buf\ \&\ < TT.active.position > \\
\quad \wedge mpath' = mpath\ \&\ < sTT.active.position > \\
\quad \wedge impath' = impath\ \&\ < TT.active.position > \\
\quad \wedge Unchanged_{Get\_New\_Pos}
\end{array}
\tag{B.10}
$$

$$
\begin{array}{l}
Deactivate\_Telegrip \doteq \\
\exists RTS, TT : \\
\quad TT.active \\
\quad \wedge TT.inactive' \\
\quad \wedge if\ (RTS.oper \vee RTS.starting)\ then \\
\quad\quad RTS.stopping' \\
\quad endif \\
\quad \wedge Unchanged_{Deactivate\_Telegrip}
\end{array}
\tag{B.11}
$$

$$
\begin{array}{l}
Start \doteq \\
\exists RTS, TT : \\
\quad TT.active \wedge RTS.not\_op \\
\quad \wedge RTS.starting' \\
\quad \wedge RTS.session' = RTS.session + 1 \\
\quad \wedge Unchanged_{Start}
\end{array}
\tag{B.12}
$$

$$
\begin{array}{l}
Com\_Start \doteq \\
\exists LTS, RTS : \\
\quad RTS.starting \wedge LTS.not\_op \wedge RTS.seq = LTS.seq \\
\quad \wedge LTS.starting' \\
\quad \wedge LTS.session' = RTS.session \\
\quad \wedge Unchanged_{Com\_Start}
\end{array}
\tag{B.13}
$$

$$
\begin{array}{l}
Release\_Hold \doteq \\
\exists LTS, R : \\
\quad LTS.starting \\
\quad \neg(R.emergency\_stop \vee R.external\_hold \vee R.alarm) \\
\quad R.still.signaled \\
\quad \wedge R.command\_hold' = false \\
\quad \wedge Unchanged_{Release\_Hold}
\end{array}
\tag{B.14}
$$

$$
\begin{array}{l}
Start\_Camera \doteq \\
\exists CM, LTS : \\
\quad LTS.camera\_idle \wedge LTS.starting \\
\quad \wedge LTS.camera\_detecting' \\
\quad \wedge CM.detecting' \\
\quad \wedge Unchanged_{Start\_Camera}
\end{array}
\tag{B.15}
$$

$$
\begin{array}{l}
Send\_Pos \doteq \\
\exists TS, sTT, sR, LTS, RTS, TT, R : \\
\quad sTT.active \wedge TS.inoperative \wedge sR.still \\
\quad \neg(sR.external\_hold \vee sR.emergency\_stop \vee sR.alarm) \\
\quad LTS.starting \wedge LTS.camera\_detecting \\
\quad R.still.idle \wedge RTS.starting \wedge TT.active \\
\quad \wedge TS.buf' = <> \\
\quad \wedge if\ TS.inoperative.camera\_off\ then \\
\qquad sCM.detecting' \\
\quad endif \\
\quad \wedge TS.operative' \\
\quad \wedge TS.operative.last\_pos' = sR.position \\
\quad \wedge sTT.active.position' = sR.position \\
\quad \wedge mpath' = < sR.position > \\
\quad \wedge rpath' = < sR.position > \\
\quad \wedge RTS.buf' = < (-RTS.session) > \\
\quad \wedge RTS.abp' = LTS.abp \\
\quad \wedge RTS.oper' \\
\quad \wedge RTS.oper.last\_pos' = R.position \\
\quad \wedge TT.active.position' = R.position \\
\quad \wedge impath' = < R.position > \\
\quad \wedge irpath' = < R.position > \\
\quad \wedge Unchanged_{Send\_Pos}
\end{array}
\tag{B.16}
$$

$$
\begin{array}{l}
Wait\_Confirm \doteq \\
\exists LTS : \\
\quad LTS.starting \wedge head(LTS.buf) \neq -LTS.session \\
\quad \wedge LTS.buf' = tail(LTS.buf) \\
\quad \wedge Unchanged_{Wait\_Confirm}
\end{array}
\tag{B.17}
$$

$$
\begin{array}{l}
Start\_Lts \doteq \\
\exists LTS : \\
\quad LTS.starting \wedge head(LTS.buf) = -LTS.session \\
\quad \wedge LTS.oper' \\
\quad \wedge LTS.buf' = tail(LTS.buf) \\
\quad \wedge Unchanged_{Start\_Lts}
\end{array}
\tag{B.18}
$$

$$
\begin{array}{l}
Operate \doteq \\
\exists LTS, R, TS, sR : \\
\quad LTS.oper \wedge LTS.buf \neq <> \wedge R.still \\
\quad \neg(R.emergency\_stop \vee R.external\_hold \vee R.alarm) \\
\quad TS.operative \wedge sR.still \\
\quad \neg(sR.emergency\_stop \vee sR.external\_hold \vee sR.alarm) \\
\quad \wedge sR.moving' \\
\quad \wedge sR.moving.destination' = head(TS.buf) \\
\quad \wedge \neg sR.command\_hold' \\
\quad \wedge TS.buf' = tail(TS.buf) \\
\quad \wedge rpath' = rpath \ \& \ head(TS.buf) \\
\quad \wedge R.moving' \\
\quad \wedge R.moving.destination' = head(LTS.buf) \\
\quad \wedge \neg R.command\_hold' \\
\quad \wedge LTS.buf' = tail(LTS.buf) \\
\quad \wedge irpath' = irpath \ \& head(LTS.buf) \\
\quad \wedge Unchanged_{Operate}
\end{array}
\tag{B.19}
$$

$$
\begin{array}{l}
Tx\_Pkt \doteq \\
\exists LTS, RTS : \\
\quad RTS.buf \neq <> \wedge LTS.abp = RTS.abp \\
\quad \neg(RTS.not\_op \vee RTS.stopping \\
\qquad \vee LTS.not\_op \vee LTS.stopping) \\
\quad \wedge LTS.buf' = LTS.buf \ \& < head(RTS.buf) > \\
\quad \wedge LTS.abp' = not(LTS.abp) \\
\quad \wedge Unchanged_{Tx\_Pkt}
\end{array}
\tag{B.20}
$$

$$
\begin{array}{l}
Ack \doteq \\
\exists LTS, RTS : \\
\quad RTS.buf \neq <> \wedge LTS.abp \neq RTS.abp \\
\quad \wedge RTS.buf' = tail(RTS.buf) \\
\quad \wedge RTS.abp' = not(RTS.abp) \\
\quad \wedge Unchanged_{Ack}
\end{array}
\tag{B.21}
$$

$$
\begin{array}{l}
Stop \doteq \\
\exists RTS : \\
\quad RTS.oper \vee RTS.starting \\
\quad \wedge RTS.stopping' \\
\quad \wedge Unchanged_{Stop}
\end{array}
\tag{B.22}
$$

$$
\begin{array}{l}
Ack\_Signal \doteq \\
\exists LTS, R : \\
\quad R.still.signaled \wedge (RTS.oper \vee LTS.starting) \\
\quad \wedge LTS.stopping' \\
\quad \wedge Unchanged_{Ack\_Signal}
\end{array}
\tag{B.23}
$$

$$
\begin{array}{l}
Motion \doteq \\
\exists CM, LTS: \\
\quad CM.detecting \\
\quad \wedge CM.idle' \\
\quad \wedge LTS.camera\_idle' \\
\quad \wedge if\ (LTS.operorLTS.starting)\ then \\
\qquad LTS.stopping' \\
\quad endif \\
\quad \wedge Unchanged_{Motion}
\end{array}
\tag{B.24}
$$

$$
\begin{array}{l}
Inform\_Not\_Op \doteq \\
\exists LTS, RTS, TT, CM, TS, sTT, sCM, sR: \\
\quad LTS.not\_op \\
\quad (RTS.oper \vee RTS.stopping \vee \\
\qquad (RTS.starting \wedge LTS.seq \neq RTS.seq)) \\
\quad \wedge RTS.not\_op' \\
\quad \wedge RTS.seq' = LTS.seq \\
\quad \wedge if\ (TS.operative \wedge sTT.active) \wedge TT.inactive\ then \\
\qquad sTT.inactive' \\
\qquad sCM.idle' \\
\qquad sR.still.signaled' \\
\qquad sR.command\_hold' \\
\qquad TS.inoperative.camera\_off' \\
\quad else\ if\ TS.operative \wedge \\
\qquad ((sTT.inactive \wedge TT.inactive) \vee \\
\qquad (sTT.active \wedge TT.active)) \\
\quad then \\
\qquad sCM.idle' \\
\qquad sR.still.signaled' \\
\qquad sR.command\_hold' \\
\qquad TS.inoperative.camera\_off' \\
\quad else\ if\ sCM.detecting \\
\qquad ((sTT.inactive \wedge TT.inactive) \vee \\
\qquad (sTT.active \wedge TT.active)) \\
\quad then \\
\qquad sCM.idle' \\
\qquad sR.still.signaled' \\
\qquad sR.command\_hold' \\
\qquad TS.inoperative.camera\_off' \\
\quad else\ if\ (TS.operative \wedge \\
\qquad (sR.external\_hold \vee sR.emergency\_stop \vee sR.alarm)) \wedge \\
\qquad ((sTT.inactive \wedge TT.inactive) \vee \\
\qquad (sTT.active \wedge TT.active)) \\
\quad then \\
\qquad TS.inoperative.camera\_on' \\
\quad else\ if\ (TS.inoperative \wedge sTT.active) \wedge \\
\qquad TT.inactive \wedge (\neg(CM.idle \vee sCM.idle)) \\
\quad then \\
\qquad sTT.inactive' \\
\quad endif \\
\quad \wedge Unchanged_{Inform\_Not\_Op}
\end{array}
\tag{B.25}
$$

$$
\begin{array}{l}
Comm\_Stop \doteq \\
\exists LTS, RTS : \\
\quad RTS.stopping \wedge \neg(LTS.stopping \vee LTS.not\_op) \\
\quad \wedge LTS.stopping' \\
\quad \wedge Unchanged_{Comm\_Stop}
\end{array}
\tag{B.26}
$$

$$
\begin{array}{l}
Rest \doteq \\
\exists LTS, R : \\
\quad LTS.stopping \wedge R.still \\
\quad \wedge LTS.not\_op' \\
\quad \wedge LTS.seq' = not(LTS.seq) \\
\quad \wedge Unchanged_{Rest}
\end{array}
\tag{B.27}
$$

$$
\begin{array}{l}
Stop\_R \doteq \\
\exists LTS, R : \\
\quad LTS.stopping \\
\quad \wedge R.still.signaled' \\
\quad \wedge R.command\_hold' \\
\quad \wedge Unchanged_{Stop\_R}
\end{array}
\tag{B.28}
$$