Process Models for Distributed Event-Based Systems

by

Rolando M. Blanco

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2010

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Distributed Event-Based Systems (DEBSs) are middleware supporting the interaction of publisher and subscriber components via events. In DEBSs, the subscribers to be notified when an event is announced are decided at run-time without requiring publisher components to know the name or locations of the subscribers, nor the subscribers to know the name or locations of the publishers. This low coupling between components makes DEBSs suitable for applications with a large or unpredictable number of autonomous components.

The development of applications in DEBSs is an ad hoc process poorly supported by current software engineering methodologies. Moreover, the behaviours exhibited by these systems and their applications are not well understood, and no suitable models exist where these behaviours can be described and analyzed. The main concern of this thesis is the development of such models. Specifically, we develop formalisms and models supporting the specification, prediction, and validation of the behaviour exhibited by the middleware and the applications executing on it.

Our main contributions to the area are: new formalisms for the representation of DEBSs and their applications, and for the specification of both, system and application properties; a categorization of the features related to the definition, announcement, and notification of events in DEBSs and, in general, event-based systems; models representing the categorized DEBS features; case studies detailing models and properties for specific systems; a prototype tool for the verification of DEBSs and applications. The formalisms developed expose the location of the actions in the modelled systems and support the specification of several forms of location-awareness and adaptive behaviour.

Acknowledgements

I would like to thank Paulo Alencar and Donald Cowan for giving me the opportunity to pursue my PhD even though it had been eleven years since I had completed my Masters. As my supervisor, Paulo gave me the freedom to follow my research interests and his advise allowed me to tackle what, in many occassions during the development of this thesis, seemed to be insurmountable problems.

I would also like to thank Ashraf Aboulnaga, Daniel Berry, Kostas Kontogiannis and Hausi Müller for their work as part of my thesis committee. Besides having them in my committee, I had the fortune of being a student of Ashraf and Daniel.

I would like to express my gratitude to my parents Germán and Consuelo, and brothers Luis Adolfo and Mauricio for their love and support. I had the privilege of growing up in a family where scholarship is encouraged and appreciated.

Finally, I am most thankful to my wife Esperanza, and our daughters Silvia and Mariana. It is because of their love, sacrifices and support that I am able to present this work.

¡Estudie y triunfe!

Dedication

This thesis is dedicated to Esperanza, Silvia, and Mariana.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The term *implicit invocation* is used to specify a software architectural style. A system follows the implicit invocation architectural style when the system has components that *announce* and *react to* events. Such a system is called an *Implicit Invocation System (IIS)*. Depending on the IIS, a component can be a module, class, or program. The functionality executed by a component reacting to an event is considered functionality that has been implicitly invoked by the announcement of the event [74, 55, 125]. Implicit invocation is generally used when a system needs to react to changes in the environment or within the system itself.

The term *reactive system* is also frequently used to refer to an IIS. Each IIS can be differentiated based on the event model it supports. The *event model* of an IIS determines how each event is defined and announced to other components, how a component specifies its interest in events, and how each event is delivered to interested components.

In this thesis we focus on Distributed Event-Based Systems (DEBSs), a particular kind of IIS. A DEBS is middleware that allows autonomous components to interact via events. A component that announces an event is called a *publisher* (cf. Figure 1.1). A component interested in an event that has been announced is called a *subscriber*. A subscriber is notified of only those events for which it has previously expressed interest.

The operations used by each component to interact with the DEBS are collectively known as the *DEBS API* [133]. In this thesis, we refer to each application implemented by publisher and subscriber components using the DEBS API as a *DEBS application*. Sometimes, when it is clear by its context, we use the term DEBS to refer to both, the middleware and the applications running on it.

The following event model features differentiate a DEBS from other IISs:

- The kinds of events, components publish and subscribe to, are not predetermined by

the system. New kinds of events can be introduced to the system, and existing kinds of events can be removed from the system at run time.

- Each component must register its interest in the events it wants to be notified about by invoking a `subscribe` operation.
- A component publishes an event by explicitly invoking an `announce` or `publish` operation.
- An event binding determines which components are to be notified when an event is announced. In each DEBS the binding between an event and the components reacting to the event can be established and terminated dynamically. Moreover, each event binding is maintained by the DEBS, without the need for a publisher component to be aware of which components are interested in its events. Hence, the publisher of an event does not specify the components that will be notified of the event.
- When an event is published, the component publishing the event continues its execution without waiting for subscriber components to be notified of the event.
- The system attempts to deliver an event to all the components interested in the event.

Due to these event model features, components in a DEBS exhibit *time*, *space*, and *synchronization decoupling* [61]. Time decoupling occurs because components do not need to be actively participating in the interaction at the same time; space decoupling occurs because components do not need to know each other for them to interact; and synchronization decoupling occurs because when a publisher announces an event, the publisher does not wait for the event to reach interested components or for their reactions to the event.

The reduced coupling between components in a DEBS allows the development of applications by integrating functionalities implemented by components that are autonomous and heterogeneous. Such applications are expected in ubiquitous computing environments [166], as well as mobile and web environments integrating a large, and sometimes unpredictable, number of participants and applications [76, 140]. Areas in which DEBSs have been applied include health informatics [152, 101, 77], mobile systems [46, 111, 113], and monitoring and steering of business, agricultural, and industrial processes [94, 148, 83].

1.1 Open Problems

Most of the research efforts in DEBSs have been oriented towards the development of commercial and prototype systems, i.e., the middleware, that can efficiently provide the functionality required for the execution of DEBS applications [128, 47, 38, 119, 20, 134].

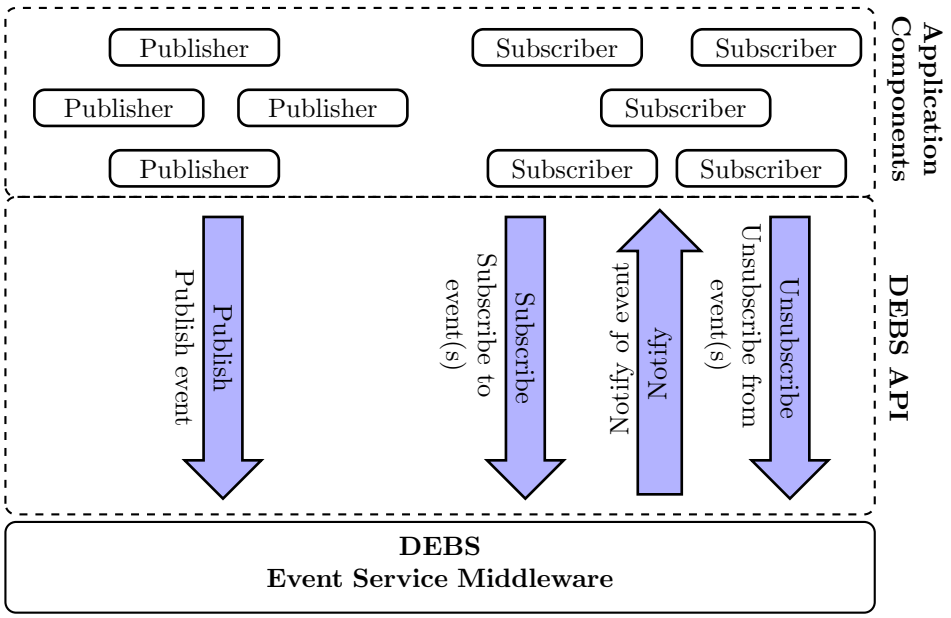


Figure 1.1: Interactions in a Simple DEBS

While formalisms and abstractions for modelling, structuring, information hiding, and modularization exist for traditional systems, equivalent formalisms and abstractions have not been proposed for DEBSs. Hence, the engineering of applications in DEBSs is still an ad hoc process poorly supported by current software engineering methods [64, 67, 122, 84].

Consider modelling formalisms typically proposed for IISs when applied to DEBSs ([e.g., 169]). These formalisms assume event models that are incompatible or do not support all the features in the DEBS event model. For example, Harel statecharts assume instantaneous event processing: the reaction to an event occurs in zero time, upon notification of the event [78]. This is not the case when dealing with DEBSs, where events take time to reach subscribed components. Another issue is the assumption that only one event may happen at a time. UML statecharts do not have this assumption, providing instead a queue of events [5]. Both Harel and UML statecharts assume broadcasting of events, where each event is globally visible to all components in the system. In contrast, in DEBSs each event is only notified to subscribed components. Because these incompatibilities between the DEBS event model and the statechart event model the use of statecharts to model behaviour in DEBSs and DEBS applications is inconvenient. This problem is not unique to DEBSs and arises when dealing with any IISs that has an event model incompatible with the statechart event model. The effect of the incompatibility between event models has been the proposal of multiple statechart variations when attempting their use to model

complex IISs ([e.g., 163, 95, 54, 17, 142]).

Other formalisms based on finite state machines for representing system behaviour also suffer from event model incompatibilities when applied to IISs ([e.g., 30]). More recently, interface automata have been used to describe the behaviour of reactive systems [51, 159, 162]. As with statecharts, the event model assumed in interface automata differs from the event model found in DEBSs. For example, in interface automata the arrival of an event while in a state not prepared to handle the event indicates an incompatibility between the environment and the automaton. In contrast, in DEBSs an event is queued until the component is at a state ready to handle the event.

In order to improve the support provided by software engineering methodologies and practices for the development and maintenance of applications in DEBSs it is critical to have a sound understanding of the DEBS event model and the relationships and behaviours exhibited in DEBSs. An approach to gaining this understanding is to study deployed DEBS applications. This approach requires the ability to instrument the applications themselves and, possibly, the DEBSs. Moreover, instrumentation requires low-level knowledge of the application code as well as access to proprietary applications, which may not be possible to obtain. An alternate approach is to develop models for DEBSs and applications. To be useful in improving our understanding of DEBSs, these models should support the specification, prediction and verification of the behaviour exhibited by the DEBSs and the applications running on them. *The development of these models and reasoning capabilities is the main concern of this thesis.*

There have been some attempts at modelling DEBSs and specifying their behaviour. Most of the prior works in the area are based on trace semantics [119, 69, 16], and a few on model checking tools [72, 32].

With trace semantics, an execution trace records the operations performed by components in a system. The scope of the work in the area has been on specifying basic liveness and safety properties. Each property specifies the types of traces that a system complying with the specification must exhibit. For example, assuming publish and notify operations, a property may establish that a notify operation for an event must not occur before the publish operation for the same event.

Modelling behaviour with traces has several shortcomings: systems must be instrumented to obtain traces; in a distributed system it can be difficult to maintain a central trace, in particular if there is no central clock guaranteeing operations are recorded in the trace in the same order they are executed; a trace exhibits the operations of a single execution and, even with multiple executions, there is no guarantee all possible traces for the system may be produced; analysis and property specification are limited to the operations recorded in the traces. A major problem with traces is that analysis occurs after execution: the system, or at least a prototype, needs to be built in order to collect traces.

We are not aware of any DEBS models in the literature supporting the specification of general application behaviour (i.e., the behaviour exhibited by publisher and subscriber components). In the case of Garlan et. al., [72], the work is restricted to outlining an approach to model checking DEBSs. A DEBS, where events may not be delivered in the same order they are announced, is modelled as a set of events in Buschmann et al. [32]. A single property, specifying that applications must not rely on the order the events are announced is then verified; specification of other properties, such as the absence or occurrence of events is not supported in that work.

1.2 Approach

Our approach for the development of models and reasoning capabilities for the specification, prediction and verification of the behaviour exhibited by DEBSs and DEBS applications is illustrated in Figure 1.2, and further described below.

Characterization of the DEBS Event Model. We start by categorizing the event-related features that make a DEBS different from other IISs. As part of the categorization we identify the features commonly found in each DEBS and the features where variations exists among different DEBSs. The categorization elicits relevant behaviour that needs to be supported when modelling a DEBS and specifying related properties.

System Representation. We introduce *kell-m*, a new process algebra for modelling DEBSs and their applications. Using kell-m we represent basic control and modularization constructs and introduce syntactic sugar with the objective of improving the readability of the models when compared to standard kell-m. The resulting language is called *hl-kell-m*, for *high-level kell-m*.

A process algebra is a formalism for the specification and description of distributed, and in general, concurrent systems [15]. The use of process algebras had previously been suggested for DEBSs [64], but we are not aware of further research taking place. Prior to kell-m we used the asynchronous π-calculus, a process algebra for the representation of asynchronous systems [87, 25]. Although the asynchronous π-calculus has few basic constructs and support for property specification and verification, the resulting models were laborious to specify and read. Typically, it is not possible to identify what behaviour was being been modelled by looking at the π-calculus code. Hence, we studied process algebras with support for localities and higher-order expressions with the intention of using these features to structure and modularize the specifications [70, 34, 82, 149, 155, 150, 21].

A locality provides the ability to group, name, and transmit processes. A first-order process algebras supports only the transmission of data. A higher-order algebra supports the transmission of data and processes, and some higher-order algebras support the transmission of localities as well. Among the algebras considered was the Kell calculus family of process algebras [155, 150, 21]. These process algebras are derived from the asynchronous π-calculus. The localities in the Kell calculus are named *kells*. Each kell is intended to represent a physical location. The term *kell* was chosen by the creators of the Kell calculus in a loose analogy with biological cells.

As we illustrate later in Section 3.6.5, the rules governing the communication of processes in the regular Kell calculus make the specifications structurally complex. The rules guarantee communications between processes are local to a kell. This is a desirable feature when a kell represents a physical location, but not when a kell is used as structuring construct or when modelling unrestricted communication between distributed components. Hence, we relax the communication rules to facilitate the use of kells as structuring constructs in the specifications. Kell-m is the new process algebra resulting from the relaxation of the communication rules.

Prior to using process algebras we tried formalisms traditionally used for the representation of IISs. In particular we used statecharts to model DEBSs [23]. We found with statecharts we could not adequately represent DEBSs without altering the statechart semantics. This was related to the differences in the event model between statecharts and DEBSs previously mentioned. Another problem already documented by others, was the inability to encapsulate behaviour in the models using statecharts [95, 151, 164].

Property Representation. The operational semantics of a process algebra determines how the systems represented with the algebra evolve. Based on the operational semantics of kell-m we develop kμ. kμ is a new modal logic for the specification of systems modelled in kell-m. With kμ it is possible to represent properties specifying the behaviour modelled systems must exhibit as they execute. Similarly to kell-m, we introduce syntactic sugar that improves the readability of the property specifications when compared to standard kμ. The resulting properties language is called hl-kμ, for high-level kμ.

Case Studies. Based on three case studies we develop kell-m models for DEBSs. The first case study is of the DEBS API standard proposed by Pietzuch et al. [133]. The main reason for the use of the standard API is that it can be supported by well-known DEBSs with little effort [133]. With this case study we show: how event-related features previously characterized for DEBSs are supported by the model; how properties previously specified for DEBSs can still be specified; and how new properties not specifiable before can now be specified. The new properties expressible with our work impose requirements on the

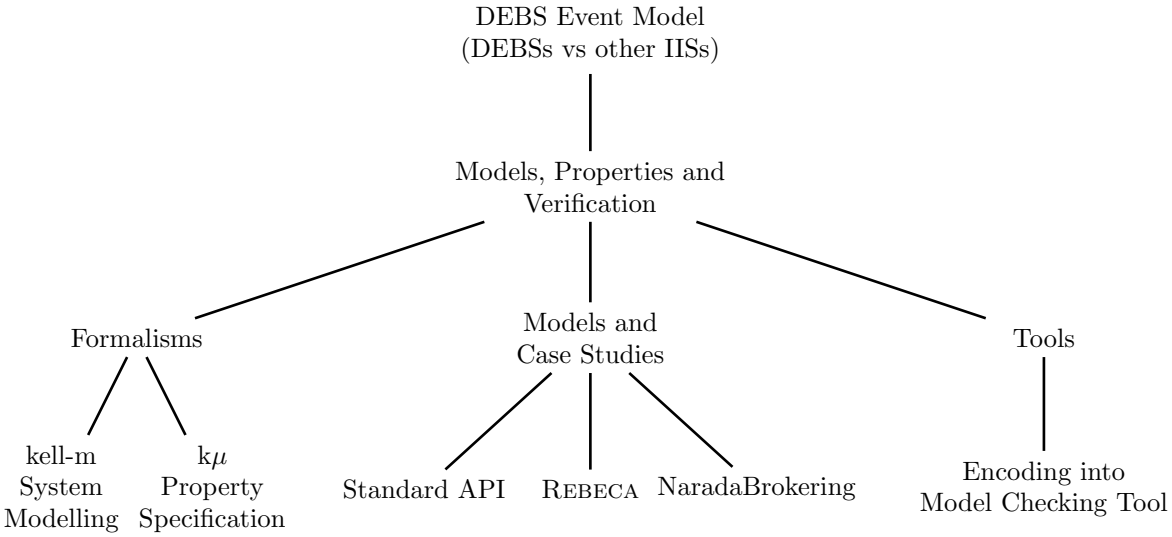


Figure 1.2: Thesis Approach

location of the actions occurring in a DEBS or its applications, and on how processing within locations adapts as the system evolves.

In the second case study, we model a hierarchical structuring mechanism for components in the Rebeca DEBS [68, 69, 67]. The model produced with this case study showcases the use of kell-m to represent a non-traditional feature proposed for DEBSs. Specifically, in Rebeca a can be grouped with other components. A group is called a *scope*, and a component can belong to multiple groups. A subscriber is only notified of events published by a component within the same scope as the subscriber.

In the final case study, we show how kells can be used to model the internal structure of administrative components in NaradaBrokering [130]. NaradaBrokering is a DEBS used in diverse applications including earthquake modelling, environmental monitoring, and video conferencing and virtual environments [10]. This case study showcases the use of kell-m to represent non-event related features in a DEBS.

Prototype Tool. We develop a prototype tool to confirm the feasibility of automating the verification of kμ properties for systems modelled in kell-m. The tool receives as input models represented in hl-kell-m and property specifications represented in hl-kμ, along with verification requests. The output of the tool is a report indicating, for each verification request, whether the model selected in the request satisfies the specification.

1.3 Contributions

Our research hypothesis is the following:

By developing formal models for DEBSs we can a) represent the behaviour of existent DEBSs, b) specify the properties that must hold as the DEBSs and their applications execute, and c) develop tools to support the automated verification of properties in DEBSs.

Our contributions in this thesis to the area of research are:

1. **Characterization of the DEBS Event Model**
   A categorization of the typical and optional features related to the definition, announcement, and notification of events that each model and, in general, formalisms and engineering methodologies should support when representing, reasoning about, and developing a DEBS or DEBS application.

2. **Formalisms for System and Property Representation**
   New process algebra kell-m, and new modal logic kμ. Kell-m is used for the representation of DEBSs and DEBS applications, kμ is used for the specification of their properties. With kell-m and kμ it is possible to express DEBS abstractions and to predict and verify behaviours exhibited by the systems modelled.

   Formalisms for property specification that have been proposed for other process algebras allow the specification of properties only based on the actions (e.g., event publication, event subscription, event notification) taken as processes evolve. With kμ it is also possible to impose conditions on the locations at which the actions take place. For example, when devices and applications are modelled with kell-m as locations, one may require a publisher of a certain kind of event to be at a specific device or application.

   The ability to localize the actions is useful, not only within the context of DEBSs, but also in other areas such as software architecture where one may be interested in identifying actions that occur within a relevant architectural location (e.g., module, class, or even an aspect in the case of Aspect Oriented Programming [90]).

   Properties specifying the stoppage and alteration of process executing within a kell can now be expressed for the first time with kμ. These properties are of interest when studying adaptation of components in DEBS applications.

   To the best of our knowledge, kμ is the only formalism that has been proposed for the specification of properties in a kell-based process algebra.

3. **Models for DEBSs**
   Models supporting the previously identified event-related features in DEBSs. These

are the first models in the area where relevant event-related features are represented. Features typically varying among different systems are parameterized in the models. A model for a specific system is produced by composing one of the models representing the functionality common to DEBSs with models detailing how the parameterized functionality is provided in the specific system.

Case studies detailing how models for specific systems are developed and how properties, both for the DEBS middleware and DEBS applications are specified.

4. **Properties for DEBSs**
   We show how properties previously identified for DEBSs can still be specified in our models. New properties, constraining the location at which the actions (e.g., event publications, event subscriptions, event notifications) take place within DEBSs and DEBS applications, and properties constraining the evolution of the systems at specific locations are also specified.

5. **Tool**
   A prototype tool for the verification of DEBSs and DEBS applications and for the study of the evolution of these systems. For the first time we are able to verify the properties that have previously been identified for DEBSs.

   To the best of our knowledge, this tool is the only tool automating the verification of properties in a kell-based algebra.

As part of the development of the formalisms and tools we have contributions that are not directly related to the engineering of DEBSs. These contributions are detailed when describing kell-m later in Chapter 3 in this thesis and are related to the reduction of higher-order process kell-m expressions to first-order kell-m expressions, semantics of kells in terms of the higher-order π-calculus, and the definition of a new bisimilarity equivalence for kell-based process algebras.

By providing models and formalisms supporting the specification, prediction and verification of both system and application behaviour in DEBSs we expect this thesis will help in advancing both the understanding and engineering of these systems.

1.4 Thesis Organization

In Chapter 2 we present the event model found in DEBSs and compare it to the event models found in other IISs. In the chapter we identify key features that need to be supported when modelling DEBSs or specifying related properties.

In the next two chapters we develop the formalisms used in the rest of the thesis. Specifically, in Chapter 3 we present kell-m, the process algebra used to represent DEBSs and DEBS applications. The focus of the chapter is on the syntax and semantics of kell-m. We illustrate how basic control and modularization constructs are represented using kell-m, and introduce a sugared syntax for kell-m that facilitates the specification and readability of the specifications. In Chapter 4 we present kμ, the modal logic used for property specification.

Using the formalisms developed in the previous chapters, in Chapter 5 we present models for DEBSs. The models are based on particular systems, standards, and features proposed by others in the area. System and application properties are specified for each of the models.

In Chapter 6 we describe a prototype model checking tool for the verification of systems represented using kell-m and properties specified in kμ. We end the thesis in Chapter 7 with conclusions and future work.

Chapter 2

Informal Event Models

In this chapter we identify and characterize the features of the DEBS event model, compare the DEBS event model to event models found in other IISs, and illustrate the variations that exist within different DEBSs. The identified features elicit the key aspects that need to be supported when modelling DEBSs or specifying related properties. The DEBS event model presented here can also be used to validate the support provided by existing software engineering methodologies for the development of applications in DEBSs, or it can be used as the starting requirement for new proposals.

Because the term *event model* is used in the area of research to characterize the event related features of a system and not to refer to a formal model of these features, in the title of this chapter we qualify the term as the *informal event model*.

2.1 Background and Chapter Organization

The term *event model* is used in the area of IISs research to characterize two different things. For some researchers, the event model of a system determines the support that the system provides for structuring event data [141]. Here, we take a more general approach, similar to that of Garlan and Scott [73], and Meier and Cahill [112]. We consider the event model to determine the application-level view that a developer must have in order to develop an event-based application or component. Hence, the event model determines how events are defined, how they are announced to other components, how components manifest their interest in events, and how events are delivered to interested components [22].

The event model found in DEBSs is frequently referred to as *publish-subscribe* [61]. The term publish-subscribe is quite generic and applies, not only to DEBSs, but to any IIS

where components invoke a `subscribe` operation to manifest their interest on events, and a `publish` operation to announce events. Since the event model of a system goes beyond the actual mechanism used to announce and subscribe to events, we refrain from using the term publish-subscribe, and refer to the event model found in DEBSs, generically, as the *DEBS event model*.

Understanding the DEBS event model is essential in the modelling and development of these systems and their applications. As previously mentioned in Chapter 1, current specification methods assume event models that are incompatible or do not support all the features in the DEBS event model.

This chapter does not provide a comprehensive list of existing DEBSs. Instead, we are interested in categorizing variations of the DEBS event model. Specific features of the DEBS event model are illustrated with a few sample DEBSs. Therefore, the reader should not assume that the mentioned DEBSs are the only systems that exhibit a particular feature.

Since there is no agreed-upon categorization of event models for IISs, to be able to compare the DEBS event model with the event models of other IISs, we introduce a general event model categorization for IISs. As discussed in Section 2.11, the categorization here presented is an extension of the categorization of implicit invocation languages proposed in [125].

Each section in this chapter covers one of the main categories: event declaration, event attributes, attribute binding, event binding, event announcement, event subscription, event delivery, event persistence, and event notification. In each section, we start by describing the category, first in terms of general IISs, and then specifically for DEBSs and their variations. For each category, a figure at the beginning of each section shows the possible variations within the category. Shaded boxes in the figures represent the variations under which the event model of an IIS must be classified to be considered a DEBSs. Figures with no shaded boxes indicate that the event model supported by a DEBS can be classified within any of the listed subcategories.

2.2 Event Declaration

The *event vocabulary* of a system is the collection of the kinds of events that components in the system can announce. Our first criterion for categorizing event models is whether or not this collection is static or dynamic and, if dynamic, whether new kinds of events need to be declared before events can be announced.

The possible variations in the event vocabulary for IISs are listed in Figure 2.1. An IIS has *fixed event vocabulary* when the kinds of events that a component can announce

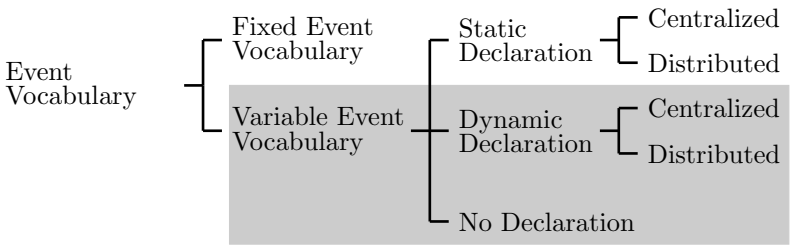


Figure 2.1: Event Declaration Variations in IISs and DEBSs

are predetermined by the system. In contrast, in an IIS with *variable event vocabulary*, different kinds of events can be added to the system as required.

When new kinds of events can be added to a system, but the set of events must be known before the system operates, the system has *static event declaration*. When new kinds of events can be added at run time, the system has *dynamic event declaration*. In some systems, there is no need to declare the kinds of events that will be announced. Such systems are said to have *no event declaration*, and events are typically announced by specifying an arbitrary string or a list of strings.

When new kinds of events need to be declared in a system, the declaration may occur at a specialized *centralized* location, or alternatively, they may occur at multiple locations. In this latter case, the system implements *distributed* declaration of events.

DEBSs commonly have variable event vocabulary: the kinds of events that can be announced by components are not predetermined by the system. Nevertheless, there are variations with regards to whether or not event types need to be declared, whether the declaration is static or dynamic, and the location of the declaration.

Before an event can be published by a component, most DEBSs require that the type of the event be registered in the system. Since the event declaration happens at run time, these DEBSs have dynamic event declaration. For example, in Hermes the components that react to events are programs [134]. Programs subscribe and unsubscribe to events via event brokers. Programs generating events register event types with the event brokers. Once an event type is registered, a program can announce events of the registered event type. Event types can be registered and unregistered dynamically.

A DEBS where new kinds of events are not declared is the Java Event-Based Distribution Architecture (JEDI) [47, 48]. Each event in JEDI corresponds to a list of strings, where the first string is the event name. The other strings in the list are the event attributes. An event subscription in JEDI specifies the name of the event and, possibly, filtering arguments on the event attributes.

A system that supports both dynamic event declarations and no event declarations is Siena [38]. Siena can operate under what the Siena creators call *subscription-based se-*

mantics and *announcement-based semantics*. Irrespectively of the operation mode, an event is a set of typed attributes. Components in SIENA are objects, and they announce events by invoking a `publish` call. Under subscription-based semantics, components interested in being notified, subscribe to events by invoking a `subscribe` call. A filtering expression specifying values for the event attributes is passed as parameter of the `subscribe` call. A component is notified of the occurrence of an event if the filtering expression specified in the subscription call matches the event notified via a `publish` call. When operating under subscription-based semantics the SIENA system requires no event declaration. In contrast, when operating under announcement-based semantics, components generating events, need to register the events they will be generating by invoking an `advertise` call. An `unadvertise` call is used by components to inform that a given event will no longer be generated. Under announcement-based semantics the SIENA system has dynamic event declaration.

With regards to the location of the declaration, an example of a DEBS where the events are declared in a central location is Yeast [93]. In Yeast events are generated when object attributes change. Yeast provides a predefined set of objects and attributes (e.g., the object `file` has attributes `file name`, `creation time`, and `modification time`). Components can declare new events by defining objects and their attributes via commands that are executed on the machine where the components run. These declarations are processed and stored by the Yeast server.

Examples of DEBSs with distributed event declaration are REBECA [119], Hermes [134], and Gryphon [20]. In these DEBSs several event brokers, possibly running at different locations in the system, process the event registration, subscription and unsubscription requests from the components generating and reacting to the events.

2.3 Event Attributes

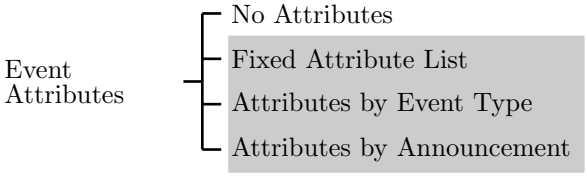


Figure 2.2: Event Attribute Variations in IISs and DEBSs

Systems may allow the association of data attributes to events. These data attributes are also known as the *parameters* of the event. If attributes are not supported, relevant information related to the event must be deduced from the event name or be retrieved,

via global variables, shared memory or database space, explicit invocation, or any other means by the component reacting to the event announcement (cf. Figure 2.2).

A system may associate the same *fixed attribute list* to each event. In most systems, events are instances of a certain *event type* or class. The term *event schema* is also used in the area to specify an event type, and in this document we use both terms interchangeably. When event types are used, the set of attributes associated with the event is determined by the type of the event. Alternatively, any number of attributes and their types may be determined at the time the event is announced. In this case, two different event announcements of events of the same type may have different parameters.

Event attributes are typically supported by DEBSs. There are variations in DEBSs with regards to whether or not all events have the same fixed attribute list. If different attributes are associated with different kinds of events, there are also variations on whether or not the attributes associated with an event are determined by the type of event.

In the Java Message Service (JMS), the same fixed attribute list is associated with each event [157]. One of the attributes in the list corresponds to a data attribute that is used by applications to associate any relevant application data to the event. Hence, although JMS-based DEBSs could be restricted by the fixed attribute list, they implement a higher level abstraction of events where the data attribute is used to provide support to typed events. A similar approach is taken when using the Common Base Event specification (CBE) [127]. CBE supports XML-based representation of events in the system. When an event is announced using a CBE representation it includes three fields: an identifier of the component announcing the event, an identifier of the component affected by the event, and relevant application data associated to the event.

Another example of a DEBS that associates the same fixed attribute list with every event is JINI [156]. JINI allows Java objects to be notified of events occurring on other, possibly remote, objects. Every event has four associated parameters: (1) an attribute identifying the type of event; (2) a reference to the object on which the event occurred; (3) a sequence number identifying the instance of the event type; (4) a hand-back object. The hand-back object is a Java object that was originally specified by the component receiving the event when the component first registered its interest on the event.

Most DEBSs require events to be typed, the event type determining the attributes in an event. For example, Hermes [134] uses XML Schema specifications [4] to represent event type definitions. The event type definitions are then used in Hermes to type check event subscriptions and publications. CAE, the Cambridge Event Architecture [14], is also a DEBS where the event attributes depend on the event type. An event occurrence is represented in CAE as the instance of a given event class. Event types are defined using an Interface Definition Language (IDL). Components interested in events of a specific class, specify a value or wildcard for each attribute of the given event class.

In DEBSs with event attributes dependent on the type of event, every single instance of a given event type has the same attributes. In contrast, JEDI [47] provides attributes by announcement. In JEDI all events are represented by a list of strings. The first string in the list indicates the event name. In JEDI there is no guarantee that two events with the same name represent the same event type. Similarly, there is no guarantee that all announcements of events of the same type are done with same-sized lists of strings.

GREEN [153] is a DEBS implemented as a framework [62] that supports different event attribute representations by way of plugins. A plugin is a specific implementation of an aspect of the event model. Plugins exist for representing events attributes as sequences of strings, name-value pairs, and objects.

2.4 Event Binding

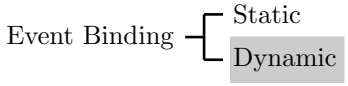


Figure 2.3: Event Binding Variations in IISs and DEBSs

The event binding in an IIS determines which components are to be informed when an event is announced. In *static event binding*, the components that react to an event are predetermined at compile time. In *dynamic event binding*, bindings between events and components that react to the events can be established or terminated dynamically (cf. Figure 2.3). Most IISs support dynamic event binding.

For example, in the Event Processing Agents and Networks approach to developing IISs proposed by Luckham [99], applications are structured hierarchically based on activity. In Luckham's view, events represent activities in the system. Each system should then be decomposed, top down, and each event representing activities at a higher level, should be defined as sequences of events in the lower levels. This layered view of events and the activities in a system, requires the ability to relate events based on hierarchy, causality, time, and aggregation. Basic functional components are called *event processing agents (EPAs)* in Luckham's work. Each EPA has an interface that specifies the events that the component produces (*out-events*), the events that the component reacts to (*in-events*), and the behaviour of the component. The behaviour is specified by reactive rules that are triggered when the EPA is notified of events of interest (*in-events*). The reactive rules generate events and change state variables local to the component.

EPAs are explicitly linked (*composed* is the term used by Luckham) to each other, based on their interfaces: out-events in one EPA are mapped to in-events in one or more

EPAs. An event generated by an EPA is only visible to the EPAs for which their in-events have been explicitly mapped to the event. The result of the interconnections is an *event processing network (EPN)*. EPNs are dynamically created and maintained. EPAs can be added or removed on the fly, and interconnections between EPAs can be changed while the EPN is in operation.

In DEBSs, bindings between events and components that react to the events can be established or terminated dynamically. Moreover, event bindings are maintained by DEBSs without the need for publisher components to be aware of which components are interested in their events.

2.5 Attribute Binding

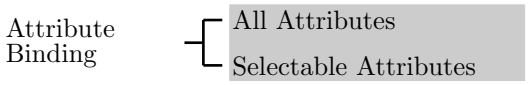


Figure 2.4: Attribute Binding Variations in IISs and DEBSs

The attribute binding in an IIS determines the attributes that are passed to an interested component when an event is announced. The majority of IISs pass *all attributes* in the event to the components reacting to the event. A different option is to allow a component to *select the event attributes* of interest (cf. Figure 2.4).

The only DEBS, of which we are aware, with support for selectable attribute binding is Padres [88]. In Padres, subscriptions are specified in a SQL-like language called PSQL. Only the event attributes listed in the select statement are delivered to the subscribed component.

2.6 Event Announcement

Most IISs have *explicit announcement* procedures that need to be used to generate an event. In some systems there is a unique *announce* or *publish* procedure or method, while in other systems there are several announcements methods (cf. Figure 2.5).

In IISs with *implicit announcement*, the event is generated as a side effect of executing an instruction or procedure. For example, in active database systems [43], components, known as database triggers, are invoked as a side effect of the insertion, deletion, or updating of data in the database.

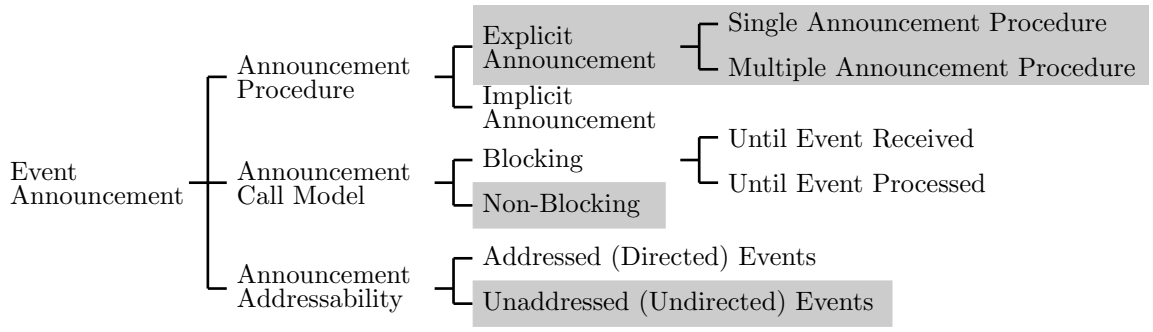


Figure 2.5: Event Announcement Variations in IISs and DEBSs

When an event is announced, the execution of the component announcing the event may be blocked *until the event is received* by all components to be notified of the event, or *until the event is processed* by all components receiving the event. Alternatively, the announcement of an event may not block the execution of the component announcing the event.

In general, centralized IISs tend to follow a blocking call model for the announcement of the event, with many of the systems blocking until the components processing the event finish their processing of the event. Most distributed IISs, on the other hand, provide a non-blocking call model.

Independently of the announcement method used by the IISs, components may or may not be required to *address* the announced events. Addressing of events, also referred to as directing of events, happens when the announcer of the event specifies the component that will be notified of the event.

Components in DEBSs explicitly announce events, the operations used to announce the events are non-blocking, and the announced events are unaddressed.

Most DEBSs have only one operation to announce events. An exception is Hermes [134], which provides three announcement methods: `publishType`, `publishTypeAttr` and `publish`. The method `publishType` announces an event that will trigger the notification of components that have subscribed to events of the given event type (or to any ancestor of the published event type, given that Hermes supports inheritance of event types). The second method, `publishTypeAttr`, triggers the notification of components that have subscribed to events of a certain type (or event type ancestor) and that have specified conditions on the attributes. Components that have subscribed to events of a given type but have not specified conditions on the event attributes are not notified of events announced using `publishTypeAttr`. The third announcement method, `publish`, has the same effect as invoking both the `publishType` and `publishTypeAttr` calls.

2.7 Event Subscription

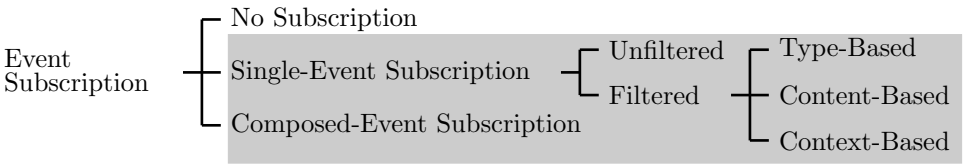


Figure 2.6: Event Subscription Variations in IISs and DEBSs

Components may or may not be required to register their interest to be notified when events are announced. If there is no requirement to register for events, also referred to as *subscribing* for events, event announcements may be broadcast to all components in the system. Alternatively, event announcements may be registered in a shared memory space that is accessed by components wishing to inquire if a certain event has been announced (cf. Figure 2.6).

Most IISs require that components register for the events about which they want to be notified. In systems with *single-event subscription*, there is a subscription procedure that must be invoked for each event of interest. In systems with *composed-event subscription*, a component can express its interest to be notified when a composition of events occurs.

Some IISs with single-event subscription allow the specification of a filtering expression as part of the subscription operation. An event is then delivered to a component only if the component is interested in the event, and the filtering expression associated with the interest of the component for the event holds. If the system supports event types, the expression can be based on the *type* of the event. Recall that when event types are supported, every generated event in the system is an instance of a certain event type. As previously mentioned, the term *event schema* is also used in the area to refer to event types.

Some systems are described as having *topic* (a.k.a. *subject*)-based subscription [61]. Events in these systems are grouped in feeds, and interested components subscribe to the feeds. We will assume that this type of filtering (topic or feed based), is an instance of type-based filtering, where a different event type can be defined for each feed of interest.

When a system allows the specification of filtering conditions on the attributes of the event, the system is said to have *content*-based filtering. An event is delivered to an interested component if the conditions imposed on the attribute values of the event are met. Recent systems also allow the filtering of events based on *contextual* information that can be related to the physical location or proximity of the components announcing and subscribed to the event [110, 153].

When IISs support composed-event subscription, a mechanism is provided for components to express event composition conditions. This mechanism is typically implemented as a language that allows the specification of temporal conditions on the event occurrences [37, 91, 97, 105]. For example, the language proposed in [91] for the specification of composite events allows the identification of a sequence of events that satisfy or violate timing and event attribute-value constraints. Based on real time logic (RTL), the specification of conditions of the type "the third occurrence of the event of type e_t after time t must have a value v for attribute $e_t.attr$" are possible in the proposed language. The existence of data repositories in the form of relational databases is assumed in [91]. Hence, conditions on the data stored in the data repositories are also part of the event composition language.

The actual event filtering may occur at a central location ([e.g., 91]), at each component ([e.g., 128]), or at specialized distributed event servers ([e.g., 38]).

DEBSs have mandatory event subscription: components must register their interest on the events they want to be notified about. Most DEBSs support single-event subscription and allow the specification of conditions on the values of the event attributes. Hence, most DEBSs support content-based event filtering. For example, in NaradaBrokering filtering conditions are specified as SQL queries on properties contained in JMS messages, as well as XPath queries [130]. In JEDI, a filtering expression is specified as an ordered set of strings [48]. Each string in the set represents a simple form of regular expression that is matched against the attribute in the same position in the event of interest. Recall that, in JEDI, an event is represented as a list of strings, where the first string in the list is the event name.

SIENA is an example of an implicit invocation system that supports composed event subscription. A filtering condition f, on the event type and event attribute values, can be specified in SIENA as part of the event subscription call. A pattern $f_1, f_2, ..., f_n$ can also be specified, where each filtering condition f_i may apply to a different event type. Such subscription indicates that the functional component running the subscription operation shall be notified if events $e_1, e_2, ..., e_n$ are generated, such that: (a) e_i occurs after e_{i-1} for all $2 \leq i \leq n$, and (b) the filtering condition f_i is true when evaluated for the event e_i, with $1 \leq i \leq n$.

GREEN supports multiple filtering models via plugins [153]. Recall that GREEN is implemented as a framework, where different aspects of the event model can have different implementations. When a GREEN system is deployed, application developers instantiate the system with the plugin implementations that meet their requirements. There is support in GREEN for type-based, content-based and proximity-based filtering. Assuming event types represented in XML, an example of a filtering condition is:

$$//RoadTraffic/[\%type = \$TrafficLight\$, colour = \$Red\$]?\#DISTANCE\# < \$100\$$$

This example (based on an example from [153]) indicates that the component should be

notified if an event of type *RoadTraffic*, representing a traffic light changing to colour *Red* within a distance of *100* units.

GREEN also supports composite-event subscription via a plugin that interfaces with CLIPS (C Language Integrated Production System) [2]. CLIPS allows the specification of Event-Condition-Action rules as filtering conditions in the event subscriptions.

2.8 Event Delivery

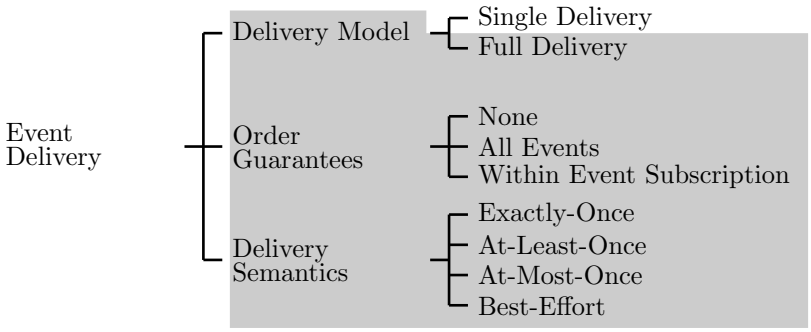


Figure 2.7: Event Delivery Variations in IISs and DEBSs

Once an event is announced, the system must select the functional components that will receive the event. In *single delivery* of events, an event is delivered to only one of the components interested in the event. Single delivery of events is useful in a pool of servers where only one server is required to attend a request represented by an event. In *full delivery*, an event is delivered to all the components interested in the event (cf. Figure 2.7). IISs implementing addressed events typically support single delivery of events.

The delivery semantics of a system determine if announced events are delivered *exactly-once*, *at-least-once*, *at-most-once*, or in *best-effort*, there are no delivery guarantees. Exactly-once and at-most-once delivery are usually more difficult to implement than the other options, since the implementation may require the use of transactional protocols. An important part of the delivery semantics, is whether or not there are order guarantees in the delivery of events. Some systems may provide order guarantees within events of a single event type. In these systems it is possible to identify, for two events of the same type, which one was generated before the other, or whether both were generated at the same time. Other systems may provide system-wide ordering guarantees. In this later case, it would be possible to identify, for any two events, even if not of the same type, which one was generated before the other.

DEBSs commonly implement full delivery of events: events are delivered to all subscribed components. There are variations with regards to the delivery semantics provided by DEBSs. For example, IBM's Gryphon project [20], implements a protocol that guarantees exactly-once delivery if the components being notified of the events maintain their connectivity to the system. The protocol models a knowledge graph where nodes, named routing brokers, represent components in charge of routing events. Arcs in the graph represent filtering conditions on the events. The filtering conditions are used to split the routing of events between routing brokers. The graph is dynamically adjusted in case of node and network failures. Further refinement of the protocol is presented in [173]. In this later work, the protocol is extended to guarantee not only exactly-once delivery, but ordered delivery of events matching a single subscription. Components, named subscription brokers, receive subscription requests from other functional components in the system wishing to be notified of events. Ordered delivery is accomplished by associating, with each event generated, a vector containing information for each subscription request related to the event. Virtual timers at subscription brokers are used to identify, from a stream of events matching a subscription, the first event in the stream after which every single event is guaranteed to be delivered, in order, to the component that subscribed to the events. To accomplish this functionality, the protocol propagates subscription information from subscription brokers to the functional components generating the events. Event information is propagated from the components generating the events to the subscription brokers. In this refinement of the original protocol, routing brokers are in charge of routing, both, the subscriptions and events.

Another system that provides delivery guarantees is NaradaBrokering [130]. Specifically, at-least-once, at-most-once, exactly-once, and ordered delivery are provided via a Web Services Reliable Messaging (WSRM) implementation [126]. For reliable delivery, the system needs to have access to reliable storage.

In JEDI, events are guaranteed to be received according to the causal relationships that hold among them [48]. In other words, if an event e_2 is generated as a reaction to another, previous event e_1, a component interested in both events is notified of e_1, before it is notified of event e_2.

2.9 Event Persistence

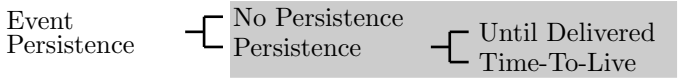


Figure 2.8: Event Persistence Variations in IISs and DEBSs

When an event is delivered, the intended recipients of the event may not be available to receive the event. In this case, the IIS may choose to save the event and attempt the delivery at a later time, or it may choose to abort the delivery of the event to the component that is unavailable. When *persistence* of events is supported by the system, the event may be maintained in the system until all intended recipients receive the event. Alternatively, the event may be maintained in the system until a *time-to-live* expires. The time-to-live may be the same for all events, it may be determined by the type of event, or it may be specified when the event is announced (cf. Figure 2.8).

Most IISs deliver events only to functional components available at the time the event is generated. The delivery semantics of the system highly influence the event persistence supported by the system. With the exception of best-effort delivery, the system must support some kind of event persistence to be able to provide delivery guarantees.

Most DEBSs deliver announced events to the subscribed components that are running at the time of the delivery. A notable exception is REBECA [119]. In REBECA, local brokers serve as access points of the system and are in charge of managing clients. Local brokers are connected to router brokers. The router brokers handle event delivery. A special kind of components, named *history clients*, save the event they are subscribed to. When a component submits a subscription request to the system, it can indicate that it wants to be notified of past events that have been saved by history clients. History clients are then notified, and they re-publish the saved events for the interested component, and only for that component. The amount of events stored by history clients is not predetermined by the system and is in general up to the history client to decide.

As previously mentioned, NaradaBrokering [130] also implements event persistence when the system has access to reliable storage. A replaying feature in NaradaBrokering allows the replaying of events at any time. This is in contrast with REBECA where replaying can happen only during the execution of a subscription operation.

2.10 Event Notification

When an event is announced, the components receiving the event may be *immediately notified* or, in *deferred notification*, notified at a later time. When the system provides immediate notification of events, the events are usually *pushed* to the components interested in the the events. The receiving components may implement a *wait loop*, or their execution may be *interrupted* by calling a previously registered callback routine (cf. Figure 2.9).

When the event notification is deferred, the component may *pull* the system for information about any events of interest that may have occurred since any previous pull call. The pull operation may be *blocking* or *non-blocking*. A hybrid push-pull option occurs

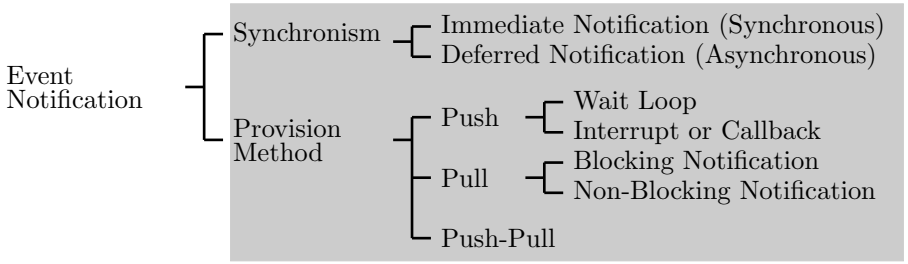


Figure 2.9: Event Notification Variations in IISs and DEBSs

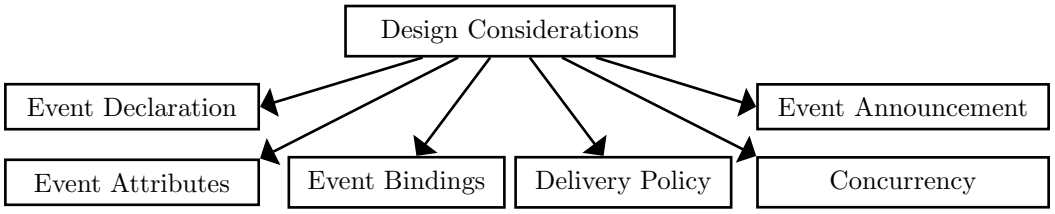


Figure 2.10: Design Considerations When Supporting Event-Based Interactions [125]

when subscribed components are signaled that an event of interest has been generated, and the components poll the system to retrieve the event data.

Most DEBSs initiate the notification actions as soon as an event is generated. The notification itself is typically pushed to the components, and a routine registered at the time of the subscription is invoked when the event is received by the component.

JEDI [48] supports, both push and pull notification mechanisms. Events are typically pushed to subscribed components, but if a subscribed component is not online when an event is generated, the event can later be retrieved by the subscribed component via a `getEvent` call. A `hasEvent` call is also used by subscribed components in pull mode to check if there are any outstanding events of interest to be processed by the component. Outstanding events are events that have been generated in the system, but have not been successfully pushed to the component.

2.11 Related Work

Although proposed for the addition of implicit invocation to traditional languages, the design considerations enumerated by Notkin et al., [125], and shown in Figure 2.10, determine the event model that will be provided by the extended language. These design considerations are:

- Event declaration: what vocabulary is used to define events, when the event definition is done, and whether or not the event vocabulary is extensible.
- Event parameters (attributes): what information is associated to events.
- Event bindings: how and when events are bound to the components that process them.
- Event announcement: whether events are announced explicitly or implicitly. If events are explicitly announced, what procedures exist to announce the events.
- Delivery policy: whether events are delivered to one or all components interested in the event.
- Concurrency: whether multiple concurrent threads of control exist in the system.

These considerations cover some of the essential properties in an event model, but important issues not related to programming languages, and relevant to distributed systems, are not included. For example, delivery semantics that arise when dealing with unreliable communication, addressability of events, and announcement call models.

Meier and Cahill take a different approach, and focus instead on the communication and structural properties of the event model in distributed IISs [112]. As shown 2.11, event models are classified as *peer-to-peer*, *mediator*, and *implicit*.

In a peer-to-peer event model, components announcing and consuming events communicate directly with each other. This form of interaction is common in systems implementing the Observer pattern [71] and Web Services Eventing (WS-Eventing) [45]. WS-Eventing is a protocol for the interaction of Web services using events. In WS-Eventing, and in most Observer pattern implementations, components that are interested in events need to know the components that announce the events, and register their interest directly with the publisher components. Publisher components must keep track of the components that are interested in the events, and when an event is announced, send the event to the interested components.

In our opinion, the peer-to-peer event model classification does not necessarily apply to DEBSs. This is because, in DEBSs, the components interested in events (similarly, announcing events) are not required to know the name nor location of the components announcing (similarly, interested in) events. The end result, is applications in DEBSs with reduced coupling when compared to Observer pattern implementations: the providers and users of functionality in DEBSs can be decided at run time, without requiring a priori knowledge of their names nor locations.

In the mediator event model, the communication is made via one or more mediator (broker) components. The mediator event model is similar to the Event Channel pattern

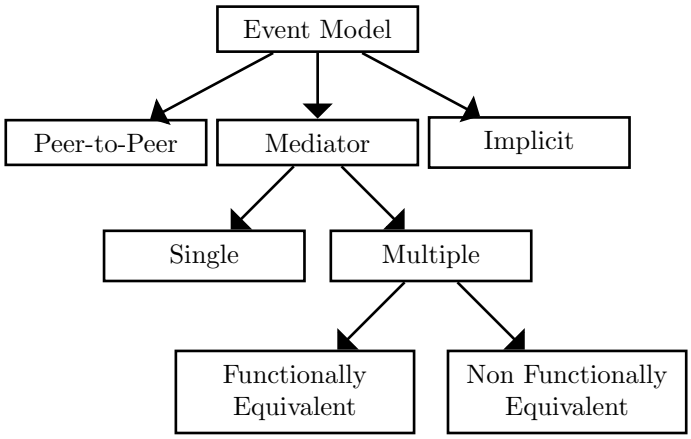


Figure 2.11: Taxonomy of DEBSs, Event Model Dimension [112]

[31], where components communicate via events without the need to register with each other, communicating instead indirectly via the mediators. Typically, different mediators are in charge of different kinds of events. Hence, components interested in a particular kind of event must be able to know and communicate with the mediator (event channel) in charge of disseminating the events of interest.

Finally, according to Meier and Cahill, in an implicit event model components consuming events subscribe to a particular event type rather than to another component or mediator. In this case the system must keep track of the components' interests and of the delivery of events. The implicit event model can be seen as a mediator event model where the system is the broker for all kinds of events.

Most of the work by Meier and Cahill focuses on the architectural aspect of the IIS, while the work of Notkin et al., focuses on some of the design options available to produce the event model for an IIS. Neither work provides a complete categorization of the diverse event models found in IISs. Notkin et al., lacks categories that are relevant in distributed IISs, while many of the categories in Meier and Cahill are defined from a structural point of view and allow features not typically found in DEBSs, for example, direct communication between publishers and subscribers.

2.12 Summary and Contributions

We presented a categorization of event models in IIS. The categorization is summarized in Figure 2.12. For each category, we discussed related features and possible variations. Unshaded categories in Figure 2.12 indicate event model features that typically do not occur in DEBSs but may occur in other IISs. The extended categorization is key to

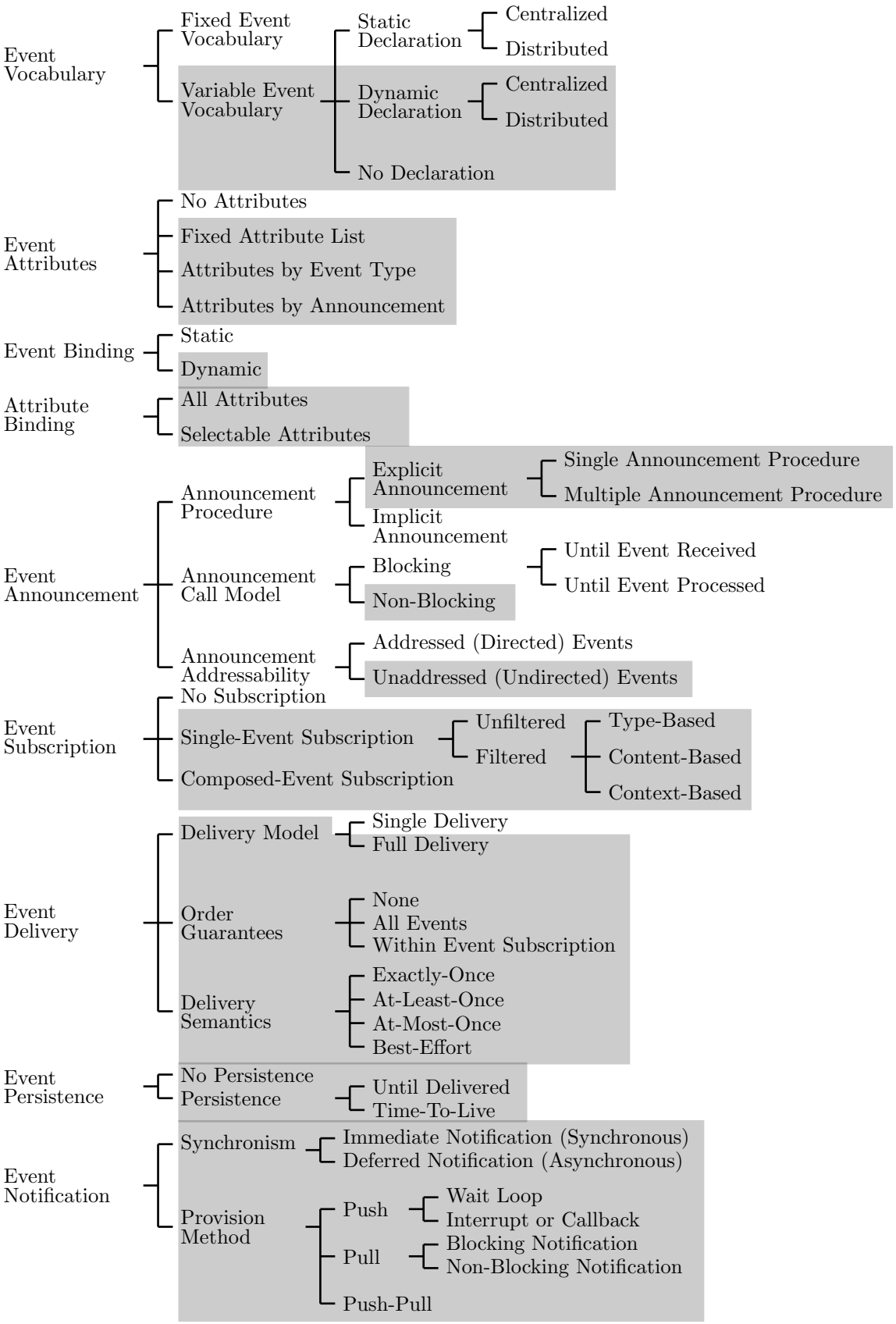


Figure 2.12: Event Model Categorization for IISs

understanding the differences between DEBSs and other IISs. The categories and typical features and variations for DEBSs are summarized below:

- *Event Declaration.* The kinds of events that components announce in a DEBS are not predetermined by the system. Some DEBSs require that publisher components register the kinds of events they will announce. The actual registration can happen at a central location, or it can happen at multiple locations within the system.

- *Event Attributes.* Data attributes can be associated to events in DEBSs. Some DEBSs allow unstructured lists of strings to be associated with the events. Other systems support attribute name-value pairs, while yet other systems support complex data structures. The actual event attributes may be determined by the type of the event, or in some systems, the attributes may vary by announcement.

- *Dynamic Event Binding.* In all DEBSs, the event bindings between publishers and subscribers can be established and terminated dynamically. The bindings are maintained by the system without the awareness of the publisher or subscriber components.

- *Attribute Binding.* In most DEBSs all event attributes are passed to subscriber components upon notification of an event. A notable exception is the Padres system [88], where subscribers specify desired attributes.

- *Event Announcement.* Publisher components must explicitly announce the events by invoking a `publish` call. Some DEBSs provide more than one publish call. The publisher component is not blocked until subscribed components receive the announced event, or until the interested components react to the event. When an event is announced, the publisher component does not address the event to any particular component.

- *Event Subscription.* Subscriber components are required to inform the system of the kinds of events of their interest via the invocation of a `subscribe` call. As part of the subscribe call, some systems allow the specification of filtering conditions. An event is then delivered to a subscribed component only if the filtering condition holds for the event. In some systems, the filtering condition, can specify multiple events. Such systems have composed event-subscription. DEBSs may support filtering conditions on the type of event, its attributes, or contextual information. The actual filtering may happen at a central location, at each component, or at specialized servers.

- *Event Delivery.* Events are delivered to all subscribed components. Order guarantees and delivery semantics vary among DEBSs. Some systems guarantee that events are delivered in order, while other systems have order guarantees only for events within

a subscription. Most systems do not provide delivery guarantees. The delivery semantics in a system specify whether an event is guaranteed to be delivered exactly-once, at-least-once, at-most-once, or if there are no guarantees.

- *Event Persistence.* Events may be kept in the system until all interested subscribers are notified, for a time-to-live period, or more frequently, the event is removed after its delivery, even if interested subscribers cannot be notified. The event persistence supported by a DEBS is typically determined by its delivery guarantees.

- *Event Notification.* When an event is announced, the system may attempt to deliver the event immediately, or it may delay its delivery. A subscriber component may be notified of an event via a previously registered call-back routine, or it may need to pull the system for any outstanding events. As part of the pull operation, if there are no outstanding events, the system may block the subscriber component until an event of interest occurs, or it may inform the subscriber of the absence of events.

The importance of this categorization for DEBSs is that it provides the typical and optional features related to the definition, announcement, and notification of events that models and, in general, formalisms and engineering methodologies should support when representing, reasoning about, and developing DEBSs and applications.

Chapter 3

Formal System Representation

In this chapter we present kell-m, a new asynchronous higher-order process algebra with hierarchical localities. Kell-m is the basis for the formal modelling of DEBSs presented in Chapter 5, and the languages and tools for verifying DEBSs and applications presented in Chapter 6.

As it is traditional when introducing a new process algebra, the focus of the chapter is on the operational semantics and behavioural equivalences for kell-m. The operational semantics determine how systems represented using kell-m evolve; the behavioural equivalences determine what it means for two kell-m processes to behave similarly.

3.1 Background and Chapter Organization

Process algebra is defined by Baeten et al. [15, p. 19-2] as:

> ⟨...⟩ the study of the behavior of parallel or distributed systems by algebraic means. It offers means to describe or specify such systems, and thus it has means to specify parallel composition. Besides this, it can usually also specify alternative composition (put things in a choice) and sequential composition (sequencing, put things one after the other). Moreover, it is possible to reason about such systems using algebra, i.e., equational reasoning. ⟨...⟩

Although, in the above definition, process algebra encompasses the whole study of distributed systems by algebraic means, the terms *process algebra* and *process calculus*, are typically used, interchangeably, just to specify the formalism used to describe and specify such systems ([e.g., 79, 115, 132]).

Kell-m is a process algebra. Computations are modelled in kell-m as processes executing in parallel. As it is traditional in process algebras, processes communicate with each other via *channels*. Channels are a kind of a more general construct, called *names*. As defined by Needham [123, p. 90], a name is:

> $\langle ... \rangle$ nothing but a bit-pattern that is an identifier, and is only useful for comparing for identity with other such bit-patterns $\langle ... \rangle$

In a communication on a particular channel, one of the processes writes to the channel and another process reads from the channel. As part of the communication, names and processes can be transmitted from the writer process to the reader process. Because processes can be transmitted in communications, kell-m is a *higher-order* process algebra.

Kell-m is derived from the Kell calculus [155, 150, 21]. The Kell calculus was created for studying component-based distributed programming. Built around a π-calculus core, the novel idea in the Kell calculus is that processes execute in localities called *kells*, where kells are identified by name. Recall the term *kell* was chosen by the creators of the Kell calculus in a loose analogy with biological cells. Kells can be dynamically created, stopped, deleted, and the process running within a kell can be replaced or composed with other processes.

Because kells are themselves processes, they can be localized within other kells, forming a tree containment hierarchy. In the traditional Kell calculus, direct communication among processes can occur only if the communicating processes are separated by at most one kell boundary. This is in contrast with kell-m, where we allow communication among processes independently of their kell location. Simplification of the requirements for process communication facilitates the use of kells as modularization constructs (cf. Section 3.6.5 for details) and matches the form in which components communicate in DEBSs. Because of this deviation from the original Kell calculus, we name our calculus the *kell-m* calculus. By convention, we do not capitalize the *k* in *kell-m*, unless we are starting a sentence.

Although the consequences of relaxing the communication rules in kell-m may be difficult to appreciate at this point of the presentation, this change makes kell-m a different formalism from the Kell calculus. It is generally the case that even small variations among process algebras produce formalisms with different semantics, properties and, ultimate, facilitate (or complicate) the use of a process algebra for representing certain kind of systems. Even more so, when the change is in the rules that control channel communication as is the case with kell-m, channel communication being one of the central aspects in process algebras.

A classic example of a small change in a process algebra producing a different formalism suitable for representing a special kind of systems is the asynchronous π-calculus [87, 25]. The main difference between the π-calculus and its asynchronous variant is that, in the asynchronous calculus, the write operation in a communication cannot be followed by any

other operation. The resulting calculus is considered asynchronous because, from the point of view of the writing process, there is no way to know when a message has been read by the receiving process [129]. Although the semantics of both calculi are close, different behavioural equivalence theories have been developed [12]. The asynchronous calculus is typically used to model computations, especially in distributed systems, where the act of sending a datum or process and the act of receiving it are separate [136].

As we show in Section 3.6, the types of constructs and systems we want to model can be represented more succinctly using kell-m that Kell calculus. Also, by not restricting the communications between processes, we show the semantics of kell-m are greatly simplified. DEBS-specific features such as the ability of components to communicate with the DEBS irrespective of their locations is directly supported by kell-m.

We start the presentation of kell-m with its syntax in Section 3.2. In Section 3.3, we introduce syntactic sugar for kell-m and show how basic control and modularization constructs can be represented using kell-m. We formalize the operational semantics of kell-m in Section 3.4 and behavioural equivalences in Section 3.5. A behavioural equivalence is used to determine if two processes have the same behaviour. We present two behavioural equivalences: barbed bisimilarity and bisimilarity up to kell containment. Barbed bisimilarity was originally proposed by Davide Sangiorgi and Robin Milner for higher-order process algebras [117], and can be applied to kell-m because the kind of operational semantics introduced in Section 3.4, and the ability to specify a predicate that detects, for a kell-m process, the possibility of performing an action. Bisimilarity up to kell containment is a behavioural equivalence, specific to kell-m, used to determine if two processes have the same kell structure. Finally, we discuss related work in Section 3.6, and summarize the chapter and list our contributions in Section 3.7.

3.2 Syntax

Every process in the kell-m calculus follows the syntax specified in Figure 3.1. P represents a kell-m process, $\mathbf{0}$ is the *null* process (a process that does not perform any actions), and $|$ is parallel composition of processes. A write action on channel a is specified $\overline{a}(\widetilde{w})$, where $\widetilde{w}$ is the sequence of names and processes being written on channel a. It is possible for a write action to transmit no values as part of the communication: $\overline{a}()$. For a channel a, when no values are passed in the channel, we sometimes write $\overline{a}$ for $\overline{a}()$. Note the calculus is asynchronous: a write operation cannot be followed by a process.

A read action is specified using *triggers*. A trigger has the form $\xi \triangleright P$, where ξ is called a *pattern*, and P is the process to execute after the read. The pattern determines the channel on which the read will occur. For example, in $a(\widetilde{c}) \triangleright P$, the pattern $a(\widetilde{c})$ specifies a read on channel a. When the read expression in a trigger pattern is matched to a write expression,

$$
\begin{array}{rcl}
P & ::= & \mathbf{0} \mid P|P \mid \overline{a}(\widetilde{w}) \mid \xi \triangleright P \mid \mathbf{new}\ a\ P \mid K[P] \mid x \mid p(\widetilde{w}) \\
\xi & ::= & a(\widetilde{v}) \mid K[x] \\
v & ::= & c \mid x \\
w & ::= & c \mid P \\
p(\widetilde{c}) & \stackrel{def}{=} & P
\end{array}
$$

Figure 3.1: Kell-m Syntax

the result of the communication is the process specified in the trigger, P in the example, with the names in $\widetilde{c}$ replaced by the values passed by the writer process. Therefore,

$$\overline{a}(d) \mid (a(e) \triangleright P) \rightarrow P\{d/e\}$$

where $P\{d/e\}$ represents the process P after all occurrences of name e have been replaced by d. For a match to occur, the number of values passed in the write expression must match the number of values expected by the read expression.

The construct **new** $a\ P$ is called *name restriction*, and it is used to specify a private name a, to be used in P. For example, assume processes:

$$P_1 : \overline{a}(d) \qquad P_2 : a(c) \triangleright R \qquad P_3 : a(e) \triangleright Q$$

When composed in parallel, P_1 can communicate with either P_2 and P_3. To guarantee the communication happens only between P_1 and P_2, a restriction on name a for P_1 and P_2 can be specified:

$$\mathbf{new}\ a\ (P_1 \mid P_2) \mid P_3$$

A private name is also called a *restricted name*. In the example, processes P_1 and P_2 are said to be within the scope of the restricted name a.

We write **new** $a,b,c\ P$ to represent **new** a **new** b **new** $c\ P$, and **new** $\widetilde{c}\ P$ to represent **new** $c_1, c_2, ..., c_n\ P$ when $\widetilde{c} = c_1, c_2, ..., c_n$.

Processes can execute inside *kells*, where a kell is just another kind of name (recall channels are also names). $K[P]$, specifies a process P running inside kell K. When $K[P]$, we say that process P is *located* in kell K. Although kells can be used to represent physical locations where the contained process execute, we also use kells to implement control and modularization constructs (cf. Section 3.3). For a given process, we refer to its *kell structure* as the containment relationships between kells in the process.

There is no restriction in the use of uppercase or lowercase letters; we typically use uppercase for kell names and processes, and lowercase for channels and process variables.

Trigger patterns can also specify kells. For example, $K[x] \triangleright R$ matches kell $K[P]$. The process variable x in R is replaced by P after the match:

$$(K[x] \triangleright R) \mid K[P] \rightarrow R\{P/x\}$$

When a kell is matched by a trigger, we say that the kell is *passivated*. Although the term *passivation* is used in the area to refer to the matching of a kell trigger and a running kell, the term is used to convey the idea that the kell being passivated is *suspended*. Hence, although kell *suspension* is probably a better term than kell *passivation*, we will use *passivation* as it is traditional in kell-based algebras.

Passivation can be used to alter or adapt the process running within a kell:

$$(K[x] \triangleright K[R|x]) \mid K[P] \rightarrow K[R\{P/x\}|P]$$

Passivation can also be used to move a process from one location to another:

$$T[P \mid K[Q]] \mid L[R \mid K[x] \triangleright K[x]] \rightarrow T[P] \mid L[R \mid K[Q]]$$

In the previous process, $K[P]$ was moved from kell T to kell L.

$p(\widetilde{w})$, represents a *process invocation*, where p is the name of the process, and $\widetilde{w}$ are the invocation parameters. The process must have been previously defined: $p(\widetilde{c}) \stackrel{def}{=} P$. An invocation $p(\widetilde{w})$ is equivalent to $P\{\widetilde{w}/\widetilde{c}\}$.

Since triggers are consumed when there is a match between a write action and the read action specified in the trigger pattern, *recurrent triggers* are introduced. A recurrent trigger is specified using $\diamond$ instead of $\triangleright$. For example:

$$(a(c) \diamond P) \mid \overline{a}(d) \rightarrow (a(c) \diamond P) \mid P\{d/c\}$$

In the previous example, the process $\overline{a}(d)$ is matched with the pattern specified by the recurrent trigger $a(c) \diamond P$. The resulting process after the match is $P\{d/c\}$, and the trigger is not consumed. Hence, the trigger can be matched to any number of write operations on channel a.

Similarly to other algebras derived from the Kell calculus, recurrent triggers are derived expressions in kell-m, since they can be expressed in terms of regular triggers [155, 150, 21]. For triggers specifying reads on channels, $\diamond$ is defined as the following *fixed point* [155]:

$$a(\widetilde{c}) \diamond P \equiv \textbf{new } t\ (Y(a, \widetilde{c}, P, t) \mid \overline{t}(Y(a, \widetilde{c}, P, t)))$$

with,

$$Y(a, \widetilde{c}, P, t) \stackrel{def}{=} t(y) \triangleright (a(\widetilde{c}) \triangleright (P \mid y \mid \overline{t}(y)))$$

Hence,

$$\begin{array}{rcl} a(\widetilde{c}) \diamond P & \equiv & \textbf{new } t\ (Y(a,\widetilde{c},P,t) \mid \overline{t}(Y(a,\widetilde{c},P,t))) \\ & \equiv & \textbf{new } t\ (t(y) \triangleright (a(c) \triangleright (P \mid y \mid \overline{t}(y))) \mid \overline{t}(Y(a,\widetilde{c},P,t))) \\ & \rightarrow & a(\widetilde{c}) \triangleright (P \mid Y(a,\widetilde{c},t) \mid \overline{t}(Y(a,\widetilde{c},P,t))) \end{array}$$

If $\overline{a}(\widetilde{d}) \mid a(\widetilde{c}) \triangleright (P \mid Y(a,\widetilde{c},P,t) \mid \overline{t}(Y(a,\widetilde{c},P,t)))$, we obtain:

$$P\{\widetilde{d}/\widetilde{c}\} \mid Y(a,\widetilde{c},P,t) \mid \overline{t}(Y(a,\widetilde{c},P,t))) \equiv P\{\widetilde{d}/\widetilde{c}\} \mid (a(\widetilde{c}) \diamond P)$$

Similarly, for trigger patterns specifying kells:

$$K[X] \diamond P \equiv \textbf{new } t\ (Y_k(K,X,P,t) \mid \overline{t}(Y_k(K,X,P,t)))$$

with,

$$Y_k(K,X,P,t) \stackrel{def}{=} t(y) \triangleright (K[X] \triangleright (P \mid y \mid \overline{t}(y)))$$

For example, a process $stop(K) \diamond (K[x] \triangleright \mathbf{0})$ receives in channel *stop* the name K of a kell; it matches the kell K's process to x, and reduces the kell to the null process $\mathbf{0}$. Such a process is useful to stop the execution of a kell:

$$\begin{array}{rl} & T[\overline{a}(b)] \mid \overline{stop}(T) \mid (stop(K) \diamond (K[x] \triangleright \mathbf{0})) \\ \rightarrow & T[\overline{a}(b)] \mid (stop(K) \diamond (K[x] \triangleright \mathbf{0})) \mid (T[x] \triangleright \mathbf{0}) \\ \rightarrow & (stop(K) \diamond (K[x] \triangleright \mathbf{0})) \mid \mathbf{0} \end{array}$$

In the previous example, the kell $T[\overline{a}(b)]$ is terminated by the process $\overline{stop}(T)$.

The symbol | has lower precedence than, both, $\diamond$ and $\triangleright$. **new** has lower precedence than $\diamond$ and $\triangleright$, but higher than |. Associativity of |, $\diamond$, and $\triangleright$ is left-to-right. For example,

$$\textbf{new } e\ a(c) \triangleright c(d) \triangleright P \mid \overline{a}(d)$$

is equivalent to:

$$(\textbf{new } e\ (a(c) \triangleright (c(d) \triangleright P))) \mid \overline{a}(d)$$

3.3 High-Level kell-m

In this section we show how basic control and modularization constructs are represented using kell-m, and introduce syntactic sugar that makes the modelling of systems less laborious. Although not DEBS-specific, these constructs will help us in the specification of DEBS functionality and makes the DEBS models more readable. We name the language resulting from the syntatic sugar *hl-kell-m*, for *high-level kell-m*. hl-kell-m is the system representation language of the prototype tool presented in Chapter 6. The grammar for hl-kell-m is specified in Appendix A.

3.3.1 Fresh Names

Although new restricted names can be created using the **new** construct, there is no corresponding construct to create new *unrestricted* names. Hence, we introduce the syntax:

$$\textbf{fresh}\ c_1, c_2, ..., c_n\ P$$

for,

$$\mathit{fresh}(c_1) \triangleright \mathit{fresh}(c_2) \triangleright \cdots \triangleright \mathit{fresh}(c_n) \triangleright P$$

and assume the existence of a process waiting on channel *fresh* for fresh name requests. Such a process returns a different name every time it is contacted. Assuming names d_1, d_2, d_3, ..., the fresh name generator is specified as follows:

$$\overline{\mathit{fresh}}(d_1) \mid \overline{\mathit{fresh}}(d_2) \mid \overline{\mathit{fresh}}(d_3) \mid \cdots$$

To guarantee the delivered name is fresh, names d_i must not occur in any other process. There must be as many writes $\overline{\mathit{fresh}}(d_i)$, as requests for fresh names are expected. If there are not enough writes, the process requesting a fresh name will be deadlocked (cf. Section 4.4, for a specification of a deadlock condition).

Fresh names can be used instead of restricted names when the newly created name should be visible by a global observer. Although not apparent at this time of the presentation, the implication of using unrestricted names is the ability to specify property conditions when representing the evolution of kell-m processes as labelled transition systems (details in Section 4.2 et seq.).

Because it is not possible to predetermine the number of required fresh names, when modelling and verifying systems represented using hl-kell-m, fresh name generation is provided by our tool (cf. Section 6.3). This is done only for efficiency since, as previously shown, a provider of fresh names can be constructed natively in kell-m.

3.3.2 Variables and Procedural Abstractions

A process for creating variables with names received on channel *vname* and values received on channel *value* can be represented as:

$$\begin{array}{l}
\mathit{var}(\mathit{vname},\ \mathit{value}) \diamond \mathit{vname}(r,\ u)\ \triangleright \textbf{new}\ R, U, c\ (\\
\quad R[\ r(rc) \triangleright \overline{\mathit{stop}}(U) \mid \overline{rc}(\mathit{value}) \mid \overline{c}(\mathit{vname},\ \mathit{value})] \mid \\
\quad U[\ u(\mathit{newval}) \triangleright \overline{\mathit{stop}}(R) \mid \overline{c}(\mathit{vname},\ \mathit{newval})] \mid \\
\quad c(n, \mathit{val}) \triangleright \overline{\mathit{var}}(n, \mathit{val}) \\
)
\end{array}$$

When composed in parallel with the previous process, $\overline{var}(v,\ a)$ makes channel v a variable with value a. A variable is then a channel that, when provided a read channel r and an update channel u, and based on which of these two channels is used, returns the value of the variable or instantiates the variable with a new value.

We introduce the following syntax for the declaration of variables; initialization values are optional:

$$\textbf{var}\ v_1 := val_i, ..., v_n := val_n$$

If no initialization value is provided, we assume the variable has a null value represented by the name *null*. The previous variable declaration is equivalent to process:

$$\overline{var}(v_1, val_1)\ |\ \cdots\ |\ \overline{var}(v_n, val_n)$$

Generic, *set* and *get* processes can be defined:

$$set(vname,\ newval) \diamond \textbf{new}\ r,u\ (\overline{vname}(r,u)\ |\ \overline{u}(newval))$$
$$get(vname,\ rc) \diamond \textbf{new}\ r,u\ (\overline{vname}(r,u)\ |\ \overline{r}(rc))$$

The following process illustrates how to retrieve, on channel *rc*, the value of a variable *vname*:

$$\overline{get}(vname,\ rc)\ |\ rc(value) \triangleright \cdots$$

By convention, if we will be using a channel to read the value returned by another process, we typically name the channel *rc*, for *return channel*.

The following type of process invocation is frequently used:

$$\textbf{fresh}\ rc\ (\overline{c}(\widetilde{ps}, rc)\ |\ rc(\widetilde{vs}) \triangleright \cdots)$$

where $\widetilde{ps}$ represents zero or more parameters written to c, and $\widetilde{vs}$ are the values returned on channel *rc*. Parameters passed can be names an processes. For these invocations we write: $@c(\widetilde{ps})(\widetilde{vs}) \triangleright \cdots$. For example,

$$\textbf{fresh}\ rc\ (\overline{get}(name,\ rc)\ |\ rc(val) \triangleright P)$$

can be written:

$$@get(name)(val) \triangleright P$$

The use of a fresh name instead of a restricted name makes the return channel visible and allows the specification of properties on actions performed on the channel (cf. Section 4.2 et seq.).

Note *val* is bound in P and no return channel is specified. A name is to be *bound* in P in it is visible to P only (a formal definition of bound names is presented in Section 3.4).

We write $@c(\widetilde{ps})$ for $\overline{c}(\widetilde{ps})$, when no channel in $\widetilde{ps}$ is used to return values. When the values returned on a channel are immediately used as inputs for another channel, for example:

$$@get(v_2)(val) \triangleright \overline{set}(v_1, val)$$

We write instead:

$$@set(v_1, @get(v_2))$$

In the previous process, we are setting the value of variable v_1 to the value stored in v_2. We further add syntactic sugar by writing this process as: $v_1 := {}^*v_2$. In general, $@set(v,\ val)$ is written $v := val$, and ${}^*v$ is syntactic sugar for $@get(v)({}^*v)$. The ${}^*v$ is just a name. Hence, if the value of a variable v is a channel, $\overline{{}^*v}(\widetilde{w})$ is valid, as well as ${}^*v(\widetilde{w}) \triangleright P$, ${}^*v(\widetilde{w}) \diamond P$, and $\overline{a}({}^*v)$.

Sometimes it is desirable to know when the value of a variable has been changed before it is read. Assuming a variable v, one could try:

$$v := newval \mid {}^*v(val) \triangleright P$$

However there is no guarantee the variable will be read after its value has been set to *newval*. Hence, we extend the definition of variables with a *synchronous update*. In a synchronous update, an extra channel uc is received along with the new value for the variable. When the update has been completed, a write is performed on channel uc:

$$\begin{array}{l} var(vname,\ value) \diamond vname(r,\ u,\ s) \triangleright \textbf{new}\ R,U,S,c\ (\\ \quad R[\ r(rc) \triangleright \overline{stop}(U) \mid \overline{stop}(S) \mid \overline{rc}(value) \mid \overline{c}(vname,\ value)] \mid \\ \quad U[\ u(newval) \triangleright \overline{stop}(R) \mid \overline{stop}(S) \mid \overline{c}(vname,\ newval)] \mid \\ \quad S[\ s(newval,\ uc) \triangleright \overline{stop}(R) \mid \overline{stop}(U) \mid \overline{c}(vname,\ newval) \mid \overline{uc}()] \mid \\ \quad c(n, val) \triangleright \overline{var}(n,val) \\ ) \end{array}$$

set and *get* are redefined as:

$$\begin{array}{l} set(vname,\ newval) \diamond \textbf{new}\ r,u,s\ (\overline{vname}(r,u,s) \mid \overline{u}(newval)) \\ get(vname,\ rc) \diamond \textbf{new}\ r,u,s\ (\overline{vname}(r,u,s) \mid \overline{r}(rc)) \end{array}$$

And a synchronous *set* is introduced:

$$syncset(vname,\ newval,\ uc) \diamond \textbf{new}\ r,u,s\ (\overline{vname}(r,u,s) \mid \overline{s}(newval,uc))$$

Then, in the process expression:

$$\textbf{new}\ uc\ (@syncset(v,\ newval,\ uc) \mid uc() \triangleright {}^*v(val) \triangleright P)$$

val is instantiated with *newval*. Introducing a synchronous assignment $:=_s$, we write the previous process expression as:

$$(v :=_s newval) \triangleright {}^*v(val) \triangleright P$$

A process P may need to wait for a variable to be created before continuing its execution, $\overline{var}(v,\ a) \mid {}^*v(b) \triangleright P$. In such cases we write:

$$\textbf{var}\ v := a\ \textbf{in}\ P$$

Finally, when defining a process, $pname(\widetilde{ps}) \stackrel{def}{=} P$, we write:

$$\textbf{process}\ pname(\widetilde{ps})\ \{\ P\ \}$$

3.3.3 Semaphores

Semaphores are used to control access to shared resources, typically variables or channels. In kell-m we model a binary semaphore as a channel on which read is possible if the resource protected by the semaphore is available. If the resource is unavailable a read is not possible, and the requesting process is blocked until the resource becomes available. A semaphore named *sem* is initialized using the write action $\overline{sem}()$. A read action $sem()$ is used to request exclusive access to the resource protected by semaphore *sem*. When the process is done with the resource, the write action $\overline{sem}()$ is invoked. For example, in:

$$\overline{sem}() \mid (sem() \triangleright P_x) \mid Q$$

The expression $\overline{sem}()$ initializes the semaphore. Process P_x has exclusive access to the resource protected by *sem* until it executes $\overline{sem}()$.

Since semaphores are modelled with channels, they can be private or global. Private semaphores are available to processes covered by the scope of the channel modelling the semaphore. Global semaphores are modelled using unrestricted channels.

Semaphores will be used when modelling DEBSs in Chapter 5 to guarantee exclusive access to variables used in the models.

3.3.4 Conditionals

Consider the following process:

$$\begin{array}{l} casetf(t, P_T, f, P_F) \diamond \textbf{new}\ T,\ F\ (\\ \quad T[t() \triangleright (P_T \mid \overline{stop}(F))] \mid \\ \quad F[f() \triangleright (P_F \mid \overline{stop}(T))] \\ ) \end{array}$$

If there is a write on channel t, then process P_T is executed. If the write is on channel f, process P_F is executed instead. The result is not predictable if there are simultaneous writes on both t and f. A process wanting to execute P_T would be:

$$\textbf{new } t,f\ (\overline{casetf}(t, P_T, f, P_F) \mid \bar{t}())$$

Similarly, to execute P_F:

$$\textbf{new } t,f\ (\overline{casetf}(t, P_T, f, P_F) \mid \bar{f}())$$

Based on this process, we write:

$$\textbf{if } @cond(\widetilde{ps}) \textbf{ then } P_T \textbf{ else } P_F \textbf{ fi}$$

to represent a process:

$$@cond(\widetilde{ps})(t,\ f) \triangleright \overline{casetf}(t, P_T, f, P_F)$$

If process P_F is $\mathbf{0}$, we write:

$$\textbf{if } @cond(\widetilde{ps}) \textbf{ then } P_T \textbf{ fi}$$

if elsif ... elsif else fi statements can be constructed using the basic **if then else fi** process.

Booleans are represented by processes at channels *true* and *false*. Each of the processes receives two channels t and f. The process waiting on *true* always writes on t; the process waiting on *false* always writes on f:

$$true(rc) \diamond \textbf{fresh } t,f\ (\overline{rc}(t, f) \mid \bar{t}())$$
$$false(rc) \diamond \textbf{fresh } t,f\ (\overline{rc}(t, f) \mid \bar{f}())$$

A process at channel *not* implements negation:

$$not(t,\ f,\ rc) \diamond \overline{rc}(f,\ t)$$

We write:

$$(\textbf{not } @cond(\widetilde{ps}))(t, f) \triangleright \cdots$$

for,

$$@cond(\widetilde{\text{ps}})(t', f') \triangleright @not(t', f')(t,\ f) \triangleright \cdots$$

This allows us to write the following expression:

$$\textbf{if } (\textbf{not } @cond(\widetilde{ps})) \textbf{ then } P_T \textbf{ fi}$$

Boolean operators **or**, **and** are similarly defined.

3.3.5 Lists

Inspired by the implementation of lists in [104, 115], a list receives two channels e and c. If the list is empty, the list writes to e, without passing any values back on e. If the list is not empty, it writes to c the list's header s and tail ss:

$$\begin{array}{l} \mathit{empty}(rc) \diamond \textbf{fresh}\ l\ (\overline{rc}(l) \mid l(e,c) \diamond \overline{e}()) \\ \mathit{cons}(s,ss,rc) \diamond \textbf{fresh}\ l\ (\overline{rc}(l) \mid l(e,c) \diamond \overline{c}(s,ss)) \end{array}$$

An empty list l is obtained by executing:

$$@\mathit{empty}()(l)$$

And a list l, with element a, is constructed by:

$$@\mathit{cons}(a, @\mathit{empty}())(l)$$

car and *cdr* are defined with their usual meaning:

$$\begin{array}{l} \mathit{car}(l,\ rc) \diamond \textbf{new}\ e,c\ (\overline{l}(e,c) \mid c(s,ss) \triangleright \overline{rc}(s)) \\ \mathit{cdr}(l,\ rc) \diamond \textbf{new}\ e,c\ (\overline{l}(e,c) \mid c(s,ss) \triangleright \overline{rc}(ss)) \end{array}$$

Also, we introduce ::, and $[;\cdots;]$ as used in OCaml [7]:

$$\begin{array}{rcl} [\,] & \equiv & @\mathit{empty}() \\ a :: [\,] & \equiv & [\,a\,] \ \equiv\ @\mathit{cons}(a, @\mathit{empty}()) \\ a :: b :: c :: [\,] & \equiv & [\,a;\ b;\ c\,] \ \equiv\ @\mathit{cons}(a, @\mathit{cons}(b, @\mathit{cons}(c, @\mathit{empty}()))) \end{array}$$

The symbol :: is therefore a shorthand for *cons*. Commas can be used instead of semicolons when specifying lists:

$$[a,b,c] \ \equiv\ [a;b;c] \ \equiv\ @\mathit{cons}(a, @\mathit{cons}(b, @\mathit{cons}(c, @\mathit{empty}())))$$

As well, we introduce:

$$\begin{array}{l} \textbf{match}\ l\ \textbf{with} \\ \quad [\,] \ \triangleright\ P_{empty} \\ \quad \textbf{or}\ s :: ss\ \triangleright\ P_{cons} \end{array}$$

to represent:

$$\textbf{if}\ @\mathit{isempty}(l)\ \textbf{then}\ P_{empty}\ \textbf{else}\ @\mathit{ht}(l)(s,\ ss) \triangleright P_{cons}\ \textbf{fi}$$

where *isempty* is defined as:

$$\begin{array}{l} \mathit{isempty}(l,\ rc) \diamond \textbf{new}\ e,\ c,\ T,\ F\ \textbf{fresh}\ t,\ f\ (\\ \quad \overline{rc}(t,\ f) \mid \overline{l}(e,\ c) \mid T[e() \triangleright \overline{t}() \mid \overline{\mathit{stop}}(F)] \mid F[c(s,\ ss) \triangleright \overline{f}() \mid \overline{\mathit{stop}}(T)] \\ ) \end{array}$$

We also specify *ht* which returns, both, the head and tail of a list:

$$ht(l,\ rc) \diamond \textbf{new}\ e\ (\bar{l}(e,rc))$$

If l is empty, $\overline{e}()$ will be written, but no process will be waiting for input on e. Hence, *ht* should only be invoked on non-empty lists.

Other usual list functionality can be represented in kell-m as follows:

$$\begin{array}{l}
foldr(p,\ v,\ l,\ rc)\ \diamond\ (\\
\quad \textbf{match}\ l\ \textbf{with}\\
\quad\quad [\,]\ \triangleright\ \overline{rc}(v)\\
\quad\quad \textbf{or}\ s :: ss\ \triangleright\ \overline{rc}(@p(s, @foldr(p,\ v,\ ss)))\\
)\\
copy(l,\ rc)kp \diamond \overline{rc}(@foldr(cons, [\,], l))\\
pos(l,\ n,\ rc)\ \diamond\ (\\
\quad \textbf{match}\ l\ \textbf{with}\\
\quad\quad [\,]\ \triangleright\ \mathbf{0}\\
\quad\quad \textbf{or}\ s :: ss\ \triangleright\ \textbf{if}\ (n = 1)\ \textbf{then}\ \overline{rc}(s)\ \textbf{else}\ \overline{pos}(ss,\ n-1,\ rc)\ \textbf{fi}\\
)
\end{array}$$

For simplicity, we assume support for numbers in kell-m. For ways to represent numbers in process algebras refer to [115].

The process at channel *del* in the following process expression deletes all occurrences of m in list l. We assume the existence of the = operator, which is able to decide if two names are the same.

$$\begin{array}{l}
append(l_1, l_2, rc)\ \diamond\ \overline{rc}(@foldr(cons, l_2, l_1))\\
reverse(l,rc)\ \diamond\ (\\
\quad \textbf{match}\ l\ \textbf{with}\\
\quad\quad [\,]\ \triangleright\ \overline{rc}([\,])\\
\quad\quad \textbf{or}\ s :: ss\ \triangleright\ \overline{rc}(@append(@reverse(ss), [s]))\\
)\\
del(l,m,rc)\ \diamond\ \overline{rc}(@foldr(d, [\,], l))
\end{array}$$

where,

$$d(s,\ l,\ rc)\ \diamond\ \textbf{if}\ s = m\ \textbf{then}\ \overline{rc}(l)\ \textbf{else}\ \overline{rc}(s :: l)\ \textbf{fi}$$

To represent sorted lists we require a channel *cmp*, that given two elements, decides if

the first element should be before the second one in the sorted list:

$$
\begin{array}{l}
cons_s(h,\ hs,\ cmp,\ rc)\ \diamond\ (\\
\quad \textbf{match}\ hs\ \textbf{with}\\
\quad\quad [\,]\ \triangleright\ \overline{rc}(@cons(h, [\,]))\\
\quad\quad \textbf{or}\ s :: ss\ \triangleright\ (\\
\quad\quad\quad \textbf{if}\ @cmp(h,s)\ \textbf{then}\ \overline{rc}(h :: hs)\ \textbf{else}\ \overline{rc}(@cons(s, @cons_s(h,\ ss)))\ \textbf{fi}\\
\quad\quad)\\
)
\end{array}
$$

A parallel *map* iterator can be implemented by:

$$
\begin{array}{l}
map(l,\ p)\ \diamond\ (\\
\quad \textbf{match}\ l\ \textbf{with}\\
\quad\quad [\,]\ \triangleright\ \mathbf{0}\\
\quad\quad \textbf{or}\ s :: ss\ \triangleright\ (\\
\quad\quad\quad @p(s)\ |\ @map(ss,\ p)\\
\quad\quad)\\
)
\end{array}
$$

We write:

$$\textbf{foreach}\ t\ \textbf{in}\ ts\ \ \textbf{do}\ P_f\ \textbf{done}$$

for,

$$\textbf{fresh}\ p\ (\ (p(t) \diamond P_f)\ |\ \overline{map}(l,\ p)\ )$$

A sequential version of the iterator can be implemented if process P_f writes to a *donec* channel when done:

$$\overline{donec}()\ |\ \textbf{foreach}\ t\ \textbf{in}\ ts\ \ \textbf{do}\ (donec() \triangleright P_f)\ \textbf{done}$$

3.3.6 Modules

We encapsulate variables and processes into *modules*, a construct similar to OO classes but without inheritance and other OO features. A module is declared with the following syntax:

$$
\begin{array}{l}
\textbf{module}\ name\ \{\\
\quad \textbf{var}\ v_1, ..., v_m;\\
\quad c_1(\widetilde{ps_1}) \triangleright P_1;\\
\quad c_2(\widetilde{ps_2}) \triangleright P_2;\\
\quad \ldots\\
\quad c_n(\widetilde{ps_n}) \triangleright P_n;\\
\}
\end{array}
$$

The module encapsulates variables $v_1, ..., v_m$, and implements operations at channels $c_1, ..., c_n$. We call these channels the *methods* of the module.

A module declaration corresponds to the following kell-m expression:

$$
\begin{array}{l}
name(rc) \diamond \textbf{fresh}\ v_1, ..., v_m\ (\\
\quad \overline{var}(v_1, null)\ |\ \cdots\ |\ \overline{var}(v_m, null)\ |\ \overline{rc}([v_1; ...; v_m]) \\
)\ | \\
name\_\_vars(self, rc)\ \diamond\ (\\
\quad @pos(self,\ 1)(v_1) \triangleright @pos(self,\ 2)(v_2) \triangleright \cdots \triangleright @pos(self,\ n)(v_n) \triangleright \overline{rc}(v_1, ..., v_n) \\
)\ | \\
name\_\_c_1(self, \widetilde{ps_1})\ \diamond\ (@name\_\_vars(self)(v_1, ..., v_n)\ \triangleright\ P_1)\ | \\
name\_\_c_2(self, \widetilde{ps_2})\ \diamond\ (@name\_\_vars(self)(v_1, ..., v_n)\ \triangleright\ P_2)\ | \\
\cdots\ | \\
name\_\_c_n(self, \widetilde{ps_n})\ \diamond\ (@name\_\_vars(self)(v_1, ..., v_n)\ \triangleright\ P_n)
\end{array}
$$

An instance of a method corresponds to an array of variables, one per variable in the method specification. A process waits on channel $name\_\_c_i$ for requests to execute method c_i. The first parameter passed is always a module instance. An extra method *name__vars* is defined for each module and returns the variables in the given array of variables. This extra method is invoked for every other module method causing references for module variables in P_i to be bounded to the variables of the given instance passed as parameter *self*.

An instance *inst* of module *name* can be created with:

$$@name()(inst) \triangleright \cdots$$

The $\triangleright \cdots$ indicates the creation of an instance is a read. Alternatively, we write:

$$inst{:}name \triangleright$$

The following process illustrates how to execute the method c_i in the newly created instance:

$$inst{:}name \triangleright @inst::name.c_i(\widetilde{ps_i})(vals) \triangleright \cdots$$

If method c_i does not return any values, the process to execute is:

$$inst{:}name \triangleright @inst::name.c_i(\widetilde{ps_i})$$

As illustrated by the previous examples, when invoking a method, the instance must be cast with the name of the module. Variable indirections $*v$ are not allowed in place of module names. Although the reason for these requirements may not be apparent at

this time, it simplifies the development of a compiler for hl-kell-m and the encoding of the algebra (cf. Chapter 6).

A module can receive parameters to be used as initial values for the variables or within the methods in the module. The syntax for a module declaration with parameters used to initialize variables is:

$$
\begin{array}{l}
\textbf{module}\ name(val_1, val_2, ..., val_m)\ \{ \\
\quad \textbf{var}\ v_1 := val_1, ..., v_m := val_m; \\
\quad c_1(\widetilde{ps_1}) \triangleright P_1; \\
\quad \cdots \\
\quad c_n(\widetilde{ps_n}) \triangleright P_n; \\
\}
\end{array}
$$

which corresponds to the kell-m process:

$$
\begin{array}{l}
name(val_1, val_2, ..., val_m, rc) \diamond \textbf{fresh}\ v_1, ..., v_m\ (\\
\quad \overline{var}(v_1, val_1)\ |\ \cdots\ |\ \overline{var}(v_m, val_n)\ |\ \overline{rc}([v_1; ...; v_m]) \\
)\ | \\
name\_\_vars(self, rc)\ \diamond\ (\\
\quad @pos(self,\ 1)(v_1) \triangleright @pos(self,\ 2)(v_2) \triangleright \cdots \triangleright @pos(self,\ n)(v_n) \triangleright \overline{rc}(v_1, ..., v_n) \\
)\ | \\
name\_\_c_1(self, \widetilde{ps_1})\ \diamond\ (@name\_\_vars(self)(v_1, ..., v_n)\ \triangleright\ P_1)\ | \\
name\_\_c_2(self, \widetilde{ps_2})\ \diamond\ (@name\_\_vars(self)(v_1, ..., v_n)\ \triangleright\ P_2)\ | \\
\cdots\ | \\
name\_\_c_n(self, \widetilde{ps_n})\ \diamond\ (@name\_\_vars(self)(v_1, ..., v_n)\ \triangleright\ P_n)
\end{array}
$$

For example, the following module *temperature* is used to store the temperature for a given latitude and longitude location. Set and get methods are specified for all variables in the module.

$$
\begin{array}{l}
\textbf{module}\ temperature(t, lt, ln)\ \{ \\
\quad \textbf{var}\ temp := t,\ lat := lt,\ lon := lt; \\
\quad gettemp(rc) \triangleright \overline{rc}(\ *temp); \\
\quad settemp(ntemp) \triangleright temp := ntemp; \\
\quad getlat(rc) \triangleright \overline{rc}(\ *lat); \\
\quad setlat(nlat) \triangleright lat := nlat; \\
\quad getlon(rc) \triangleright \overline{rc}(\ *lon); \\
\quad setlon(nlon) \triangleright temp := nlon; \\
\}
\end{array}
$$

We create a temperature instance t for a temperature of 22°C at location (43°, 80°) by invoking:

$$t{:}temperature(22, 43, 80) \triangleright \cdots$$

To avoid the need to specify set and get methods for variables in a module, these methods are always provided. Moreover, we write **(t::temperature.temp)* for:

$$@t::temperature.gettemp()(val)$$

And, $t::temperature.temp := newtemp$ for:

$$@t::temperature.settemp(newtemp)$$

Since module variables are regular variables, only get methods are required. Once a variable method has been returned by the get method, its value can be set as a regular variable. In the case of the tools described in Chapter 6, the name of the get method, as generated by the tool, is *get__varname*. The choice of name is tool dependent. Knowing the name of the method allows the specification of properties using the formalism described in Chapter 4. This is useful when one may be interested in verifying certain conditions are met when a module variable is accessed.

3.3.7 Interfaces and Module Extension

An interface specifies the methods that modules implementing the interface must provide. Interfaces do not represent executable behaviour and, as we will show in Chapter 5, are used for notational convenience to structure specifications. Behaviourally, interfaces are equivalent to the null process.

By convention, no variables are specified for interfaces and no instances of interfaces are created. To specify values returned by a method, a $\overline{rc}(\widetilde{w})$ process is specified for the method. If no values are returned, **0** is specified instead:

interface *debs* {
  subscribe(*filter, callback, rc*) ▷ $\overline{rc}$(*subscription*);
  unsubscribe(*subscription*) ▷ **0**;
  publish(*event*) ▷ **0**;
}

The interface in this example specifies basic methods in a DEBSs. The only method returning values in the interface is *subscribe*, which receives *filter*, *callback*, and *rc* channels, and returns a *subscription* channel.

A module implementing an interface is specified with the following syntax:

module *debsimpl* **implements** *debs* {
  publish(*event*) ▷ P;
  subscribe(*filter, callback, rc*) ▷ S;
  unsubscribe(*subscription*) ▷ U;
}

Where P, S, and U are the actual kell-m process expressions specifying the functionality for the corresponding methods. A module can only implement one interface. Tools processing the hl-kell-m expression should confirm that the module provides methods for all the methods specified in the interface definition and that the signatures of the methods in the module match the signature of the methods in the interface. The actual kell-m resulting from the expression is the same with and without the **implements** part.

Interfaces can be extended using the following syntax:

$$\begin{array}{l}\textbf{interface}\ \mathit{debsadv}\ \textbf{extends}\ \mathit{debs}\ \{\\ \quad \mathit{advertise}(\mathit{filter},\ \mathit{rc}) \triangleright \overline{\mathit{rc}}(\mathit{advertisement});\\ \quad \mathit{publish}(\mathit{advertisement},\ \mathit{event}) \triangleright \mathbf{0};\\ \}\end{array}$$

Extensions can provide new method specifications or override previously specified methods. In the previous example, the interface extends the *debs* interface with advertisements, and specifies different parameters for the method *publish*, which now requires the advertisement corresponding to the event being published.

Modules can be similarly extended:

$$\begin{array}{l}\textbf{module}\ \mathit{debsdiff}\ \textbf{extends}\ \mathit{debsimpl}\ \{\\ \quad \mathit{publish}(\mathit{event}) \triangleright P';\\ \}\end{array}$$

The new module *debsdiff* differs from the module it is extending in the implementation of the method *publish*. The kell-m expression resulting from the module specification for *debsdiff* is:

$$\begin{array}{l}\mathit{debsdiff}(\mathit{rc}) \diamond \overline{\mathit{rc}}([\ ])\ \mid\\ \mathit{debsdiff\_\_vars}(\mathit{self}, \mathit{rc})\ \diamond\ \overline{\mathit{rc}}()\ \mid\\ \mathit{debsdiff\_\_publish}(\mathit{self}, \mathit{event})\ \diamond\ (@\mathit{debsdiff\_\_vars}(\mathit{self})()\ \triangleright\ P')\ \mid\\ \mathit{debsdiff\_\_subscribe}(\mathit{self}, \mathit{filter}, \mathit{callback}, \mathit{rc})\ \diamond\ (@\mathit{debsdiff\_\_vars}(\mathit{self})()\ \triangleright\ S)\ \mid\\ \mathit{debsdiff\_\_unsubscribe}(\mathit{self}, \mathit{subscription})\ \diamond\ (@\mathit{debsdiff\_\_vars}(\mathit{self})()\ \triangleright\ U)\end{array}$$

Finally, a module can implement an interface and extend another module:

$$\begin{array}{l}\textbf{module}\ \mathit{modname}\ \textbf{implements}\ \mathit{internname}\ \textbf{extends}\ \mathit{basemodname}\ \{\\ \quad \ldots\\ \}\end{array}$$

Tools processing the previous hl-kell-m process expression should confirm that the module *modname*, resulting from the extension of the base module *basemodname*, provides methods

$$
\begin{array}{ll}
bn(\mathbf{0}) = \emptyset & fn(\mathbf{0}) = \emptyset \\
bn(x) = \emptyset & fn(x) = \{x\} \\
bn(\mathbf{new}\ a\ P) = \{a\} \cup bn(P) & fn(\mathbf{new}\ a\ P) = fn(P) \setminus \{a\} \\
bn(\overline{a}(\widetilde{w})) = bn(\widetilde{w}) & fn(\overline{a}(\widetilde{w})) = \{a\} \cup fn(\widetilde{w}) \\
bn(K[P]) = bn(P) & fn(K[P]) = \{K\} \cup fn(P) \\
bn(\widetilde{w}) = \bigcup_{w_i \in \widetilde{w}} bn(w_i) & fn(\widetilde{w}) = \bigcup_{w_i \in \widetilde{w}} fn(w_i) \\
bn(a(\widetilde{c}) \triangleright P) = \{\widetilde{c}\} \cup bn(P) & fn(a(\widetilde{c}) \triangleright P) = fn(P) \setminus \{\widetilde{c}\} \\
bn(K[x] \triangleright P) = \{x\} \cup bn(P) & fn(K[x] \triangleright P) = fn(P) \setminus \{x\} \\
bn(P \mid Q) = bn(P) \cap bn(P) & fn(P \mid Q) = fn(P) \cup fn(Q) \\
bn(p(\widetilde{w})) = bn(P_d\{\widetilde{w}/\widetilde{c}\}) & fn(p(\widetilde{w})) = fn(P_d\{\widetilde{w}/\widetilde{c}\}),\ \text{with } p(\widetilde{c}) \stackrel{def}{=} P_d
\end{array}
$$

Figure 3.2: Bound and Free Names in kell-m Processes

for all the methods specified in the interface definition and that the signatures of the methods in the module match the signature of the methods in the interface. The resulting kell-m expression is the same as the kell-m expression obtained from the following hl-kell-m process:

module *modname* **extends** *basemodname* {
  ...
}

This concludes our representation of control and modularization constructs using kell-m. The constructs and syntactic sugar introduced are named hl-kell-m and are used in Chapter 5 when modelling DEBSs, and are the basis for the tools presented in Chapter 6. In the next section we formalize the semantics of kell-m. The semantics determine how kell-m processes evolve as they communicate.

3.4 Operational Semantics

As it is traditional when introducing a process algebra, in this section we provide two compatible operational semantics for our kell-m calculus: labelled transition system (LTS) and reduction semantics. With LTS semantics transition rules specify the possible evolution of a process. With reduction semantics, a reduction relation determines how a process evolves as communications occur.

Independently of the operational semantics, names in process expressions can be *bound* or *free*. Recall channels and kells are names. A free name is visible to any process. A bound name is visible only to the process expression where it is bound. The functions *fn* and *bn*, defined in Figure 3.2, produce the sets of free and bound names for kell-m processes.

Communication can happen between any two processes independently of their kell location. If a bound name is output via a channel, the name becomes visible to the receiving process. The term *scope extrusion* is used to specify this situation. For example, in the process expression:

$$(\textbf{new } c\ \overline{a}(c)) \mid a(d) \triangleright P$$

there is scope extrusion because the bound name c is output via channel a to a process $a(d) \triangleright P$, where c is not bound before the process receives c.

When a name c is bound in a process P, all occurrences of c can be replaced by b, written $P\{b/c\}$, if $b \notin fn(P)$. In terms of classical process theory, c is *alpha converted* to b [115]. The behaviour of P is not affected by the alpha conversion. In Section 3.5, we formally define what it means for P and $P\{b/c\}$ to have the same behaviour but, for now, the idea is that a process Q cannot differentiate whether it is interacting with P or $P\{b/c\}$. An interaction can be either a channel communication or a kell passivation.

Structural equivalences determine the processes for which their behavioural equivalency follows immediately from their structure [132]. We write $R \equiv S$ when processes R and S are structurally equivalent. Alpha conversion is an example of a structural equivalence. The structural equivalences for the kell-m calculus are specified in Figure 3.4: parallel composition of processes is commutative and associative, and the null process is its neutral element; restriction of a name in a null process is equivalent to the null process; and the scope of name restriction can be extended to the parallel composition if the restricted name is not free in the composed process.

It is tempting to include $K[\textbf{new } c\ P] \equiv \textbf{new } c\ (K[P])$ as an structural equivalence. In [155], Stefani observed that for the Kell calculus such an equivalence introduces problems, as illustrated by the following example:

$$K[\textbf{new } c\ P] \mid (K[x] \triangleright x \mid x)$$

The resulting expression for such a process, after the kell K is passivated, is then:

$$(\textbf{new } c\ P) \mid (\textbf{new } c\ P)$$

If the structural equivalence $K[\textbf{new } c\ P] \equiv \textbf{new } c\ (K[P])$ is allowed, another, different expression, is obtained:

$$\textbf{new } c\ (K[P]) \mid (K[x] \triangleright x \mid x) \rightarrow \textbf{new } c\ (P \mid P)$$

Hence, these two ($K[\textbf{new } c\ P]$ and $\textbf{new } c\ (K[P])$) supposedly structural equivalent expressions would behave differently. Since the example is also a valid Kell calculus expression, it is also incorrect to have the equivalence in the Kell calculus. In general, this is an important issue with Kell calculus and variations of the calculus that support kell passivation, since it

$$P \mid \mathbf{0} \equiv P \qquad P \mid Q \equiv Q \mid P \qquad (P \mid Q) \mid R \equiv P \mid (Q \mid R)$$

$$P \equiv P\{\widetilde{c}/\widetilde{d}\}, \text{with } \widetilde{d} \in bn(P) \text{ and } \widetilde{c} \notin fn(P) \qquad \textbf{new } a,\ b\ P \equiv \textbf{new } b,\ a\ P$$

$$\textbf{new } a\ \ \mathbf{0} \equiv \mathbf{0} \qquad \frac{a \notin fn(Q)}{(\textbf{new } a\ P) \mid Q \equiv \textbf{new } a\ (P \mid Q)}$$

Figure 3.3: Structural Equivalences for kell-m

is desirable to be able to allow scope extrusion of names restricted within a kell. Bidinger et al. tackle this issue by disallowing the passivation of kells where processes have name restrictions [21]. Hence, before a kell can be passivated, restrictions are lifted outside the kell. This is possible by altering the semantics of the calculus and allowing processes to evolve from $K[\textbf{new } c\ P]$ to $\textbf{new } c\ (K[P])$. The main drawback in Bidinger's approach is that it restricts the semantics of kell passivations: in our example and using Bidinger's approach, it is not possible to obtain $(\textbf{new } c\ P) \mid (\textbf{new } c\ P)$ from $K[\textbf{new } c\ P] \mid (K[x] \triangleright x \mid x)$. The main advantage is a simpler implementation for the calculus.

Our approach to this problem is slightly different. In the encoding of kell-m (cf. Section 6.3.3), we do not restrict passivation of kells with name restrictions. Instead, nondeterministically, a kell can be passivated or the name restriction is extracted from the kell. Our solution has the advantage of maintaining kell passivation semantics.

3.4.1 LTS Semantics

As it is traditional in process algebras, we use a labelled transition system (LTS) to give the operational semantics of kell-m. The LTS describes the possible evolution of a process. Actions performed during the transitions can be: $\overline{a}(\widetilde{w})$, $a(\widetilde{c})$, $\overline{K}[P]$, $K[x]$, and τ:

- $\overline{a}(\widetilde{w})$ represents an output action on channel a.
- $a(\widetilde{c})$ represents an input action, via a matching trigger $a(\widetilde{c}) \triangleright R$, on channel a.
- $\overline{K}[P]$ represents an active kell with name K and process P.
- $K[x]$ represents the input of the process of kell K, via a matching trigger $K[x] \triangleright Q$.
- τ represents the matching of input and output actions on the same channel, or a kell passivation. A kell passivation corresponds to actions $\overline{K}[P]$ and $K[x]$ matching for the same kell K.

$$\frac{P \xrightarrow{\alpha} R,\ P \equiv Q}{Q \xrightarrow{\alpha} R}\ \textsc{Struct}$$

$$\overline{a}(\widetilde{w}) \xrightarrow{\overline{a}(\widetilde{w})} \mathbf{0}\quad \textsc{Out} \qquad a(\widetilde{c}) \triangleright P \xrightarrow{a(\widetilde{c})} P\quad \textsc{In}$$

$$K[P] \xrightarrow{\overline{K}[P]} \mathbf{0}\quad \textsc{KellOut} \qquad K[x] \triangleright P \xrightarrow{K[x]} P\quad \textsc{KellIn}$$

$$\frac{P \xrightarrow{\alpha} Q,\ c \notin bn(\alpha)}{\mathbf{new}\ c\ P \xrightarrow{\alpha} \mathbf{new}\ c\ Q}\ \textsc{Restrict} \qquad \frac{P \xrightarrow{\alpha} Q,\ K \notin bn(\alpha)}{K[P] \xrightarrow{\alpha} K[Q]}\ \textsc{Advance}$$

$$\frac{P \xrightarrow{\alpha} Q, bn(\alpha) \cap fn(R) = \emptyset}{P|R \xrightarrow{\alpha} Q|R}\ \textsc{Par} \qquad \frac{P_d\{\widetilde{w}/\widetilde{x}\} \xrightarrow{\alpha} Q,\ p(\widetilde{x}) \stackrel{def}{=} P_d}{p(\widetilde{w}) \xrightarrow{\alpha} Q}\ \textsc{Proc}$$

$$\frac{P \xrightarrow{\overline{a}(\widetilde{w})} Q,\ c \in names(\widetilde{w}),\ c \neq a}{\mathbf{new}\ c\ P \xrightarrow{\overline{a}(\widetilde{w}')} Q,\ \text{with } \widetilde{w}' = \widetilde{w}\{\mathbf{new}\ c\ /c\}}\ \textsc{Open}$$

$$\frac{P \xrightarrow{a(\widetilde{c})} P',\ Q \xrightarrow{\overline{a}(\widetilde{w})} Q'}{P \mid Q \xrightarrow{\tau} P'\{\widetilde{w}/\widetilde{c}\} \mid Q'}\ \textsc{L-React} \qquad \frac{P \xrightarrow{K[x]} P',\ Q \xrightarrow{\overline{K}[R]} Q'}{P \mid Q \xrightarrow{\tau} P'\{R/x\} \mid Q'}\ \textsc{L-Suspend}$$

$$\frac{P \xrightarrow{a(\widetilde{d})} P',\ Q \xrightarrow{\overline{a}(\widetilde{w})} Q',\ \widetilde{c} \subseteq \widetilde{w},\ (\mathbf{new}\ c\ ) \in \widetilde{w} \text{ if } c \in \widetilde{c}}{P \mid Q \xrightarrow{\tau} \mathbf{new}\ \widetilde{c}\ (P'\{\widetilde{w}/\widetilde{c}\} \mid Q')}\ \textsc{L-Close}$$

Figure 3.4: Labelled Transition System Semantics for kell-m

The transitions for kell-m are specified in Figure 3.4. The function *names* in the figure produces, for a process P, the set of free and bound names in P: $names(P) = bn(P) \cup fn(P)$.

Rule Struct, specifies that, if a process P transitions with action α to process R, any process Q, structurally equivalent to P, can also transition to R with action α.

Rules Out, In, KellOut, and KellIn, specify transitions due to basic communication actions. In terms of classical process theory [115], both $\overline{a}(\widetilde{w})$ and $K[P]$ correspond to

concretions. Trigger matching expressions $a(\widetilde{c})$ and $K[x]$ in triggers $a(\widetilde{c}) \triangleright P$, and $K[x] \triangleright P$ correspond to *abstractions*.

Rule RESTRICT specifies that a transition α occurs in P and **new** c P, if c, the name restricted, is not bound in the transition action. Since bn was defined for process expressions (cf. Figure 3.2), not for transition actions, here we are abusing its definition. Hence, we extend bn to deal with transition actions as follows:

$$\begin{array}{l} bn(a(\widetilde{c})) = \{\widetilde{c}\} \\ bn(K[x]) = \{x\} \\ bn(\overline{a}(\widetilde{w})) = bn(\widetilde{w}) \\ bn(\overline{K}[P]) = bn(P) \end{array}$$

Rule ADVANCE specifies that if a process P can transition to a process Q, the same process P, when within a kell K, can also transition with the same action α. This is the case when a kell is not passivated: the process located within the kell can transition (*advance*). Once the process has advanced, it can, again, be passivated or advance.

Although not obvious at this point of the presentation, the rules KELLOUT and ADVANCE provide an intuitive notion for the semantics of kells in kell-m. These two rules specify that every active kell $K[P]$ can be seen as a higher-order processes that, non deterministically, either outputs P on a channel named K, or advances in the evolution of P. In Chapter 6, we make use of this intuitive notion to encode kell-m using a variation of the π-calculus. In Chapter 6, we also show that this intuition is valid, in the sense that an observer is not able to distinguish between a kell-m process and its π-calculus encoding.

When the bound names in action α do not occur free in process R, and a process P transitions with action α to process Q, the rule PAR specifies that the process resulting from the parallel composition of P and another process R, can transition with the same action α to process $Q|R$. The condition $bn(\alpha) \cap fn(R) = \emptyset$ guarantees that free names in R are not, inadvertently, captured in an alpha conversion after a communication, as illustrated by the following sample processes:

$$P : a(c) \triangleright \mathbf{0} \qquad R : \overline{b}(c) \qquad T : \overline{a}(e)$$

Because of rule IN, $P \xrightarrow{a(c)} \mathbf{0}$, and because of rule OUT, $T \xrightarrow{\overline{a}(e)} \mathbf{0}$. If $bn(\alpha) \cap fn(R) = \emptyset$ is not required in PAR, then $P|R \xrightarrow{a(c)} \mathbf{0}|R$. Because of rule L-REACT:

$$\frac{P|R \xrightarrow{a(c)} \mathbf{0}, T \xrightarrow{\overline{a}(e)} \mathbf{0}}{(P|R) \mid T \xrightarrow{\tau} (\mathbf{0}|R)\{e/c\} \mid \mathbf{0}} \text{ L-REACT}$$

But, since $c \in fn(R)$, c is captured by the alpha conversion $(\mathbf{0}|R)\{e/c\}$, resulting in $(\mathbf{0}|\overline{b}(e))$, which is incorrect.

Processes can be parameterized using the syntax $p(\widetilde{x}) \stackrel{def}{=} P_d$. Rule PROC, deals with these definitions. The rule specifies that a process invocation $p(\widetilde{w})$ can transition to Q, if the process definition P_d, when the parameters $\widetilde{x}$ have been replaced by the values $\widetilde{w}$, can also transition to Q.

The rule OPEN deals with scope extrusion. If a restricted name c is output on an channel, the list of output values $\widetilde{w}$ is modified by replacing c with **new** c. The rule L-CLOSE specifies what happens when the restricted name is received: the name must continue to be restricted and its scope now reaches the reader and writer processes.

Rules L-REACT and L-SUSPEND specify the cases when a communication occurs on a channel, and when a passivation occurs. R-* transition rules can be trivially deduced by first using the STRUCT rule, and then the corresponding L-* rule. When illustrating transitions for process expressions, we sometimes write the name of the channel involved in a communication action right after the τ, e.g., $P \xrightarrow{\tau,a} Q$.

We refer to rules *-REACT, *-SUSPEND and *-CLOSE as the *communication rules*. These are the rules where abstractions and concretions are matched. The transitions in these rules are decorated with τ.

To illustrate the use of the transition rules, consider the process P defined as:

$$P : stop(K) \triangleright (K[x] \triangleright \mathbf{0}) \mid T[t(x) \triangleright x \mid \overline{stop}(F)] \mid F[f(x) \triangleright x \mid \overline{stop}(T)]$$

Any process received on channel t or f is executed. If channel t is used, channel f is discarded, and vice versa. As we saw in Section 3.3.4, such a process is useful to represent conditionals. In Figure 3.5, we show that $\mathbf{0} \mid \mathbf{0} \mid T[P_t|\mathbf{0}] \mid \mathbf{0}$ can be inferred from $\overline{t}(P_t) \mid P$. In Figure 3.5 we have rearranged the process expressions to facilitate the drawing of an inference tree; process expressions involved in the transitions are double-underlined.

We use the notation $K^n[P]$ to specify a kell K and its process P when the kell is embedded within $n-1$ other kells:

$$\begin{array}{ll} K^1[P] & \text{when } K[P] \\ K^2[P] & \text{when } \exists K_1 : K_1[K[P] \cdots] \\ \cdots & \\ K^n[P] & \text{when } \exists K_1, ..., k_{n-1} : K_1[K_2[\cdots K_{n-1}[K[P] \cdots] \cdots] \cdots] \end{array}$$

We call n the *depth-level* of kell K, and write $K^*[P]$ to specify a kell-m process at any depth-level. In particular, we write $K^0[P]$ when P is not within a kell.

For notational convenience, we introduce generalized versions of the communication rules *-REACT, *-SUSPEND and *-CLOSE as specified in Figure 3.6 (only the left-hand versions are shown, right-hand versions are similarly defined). The generalized rules are equivalent to applications of the PAR and ADVANCE rules, before using the communication rules.

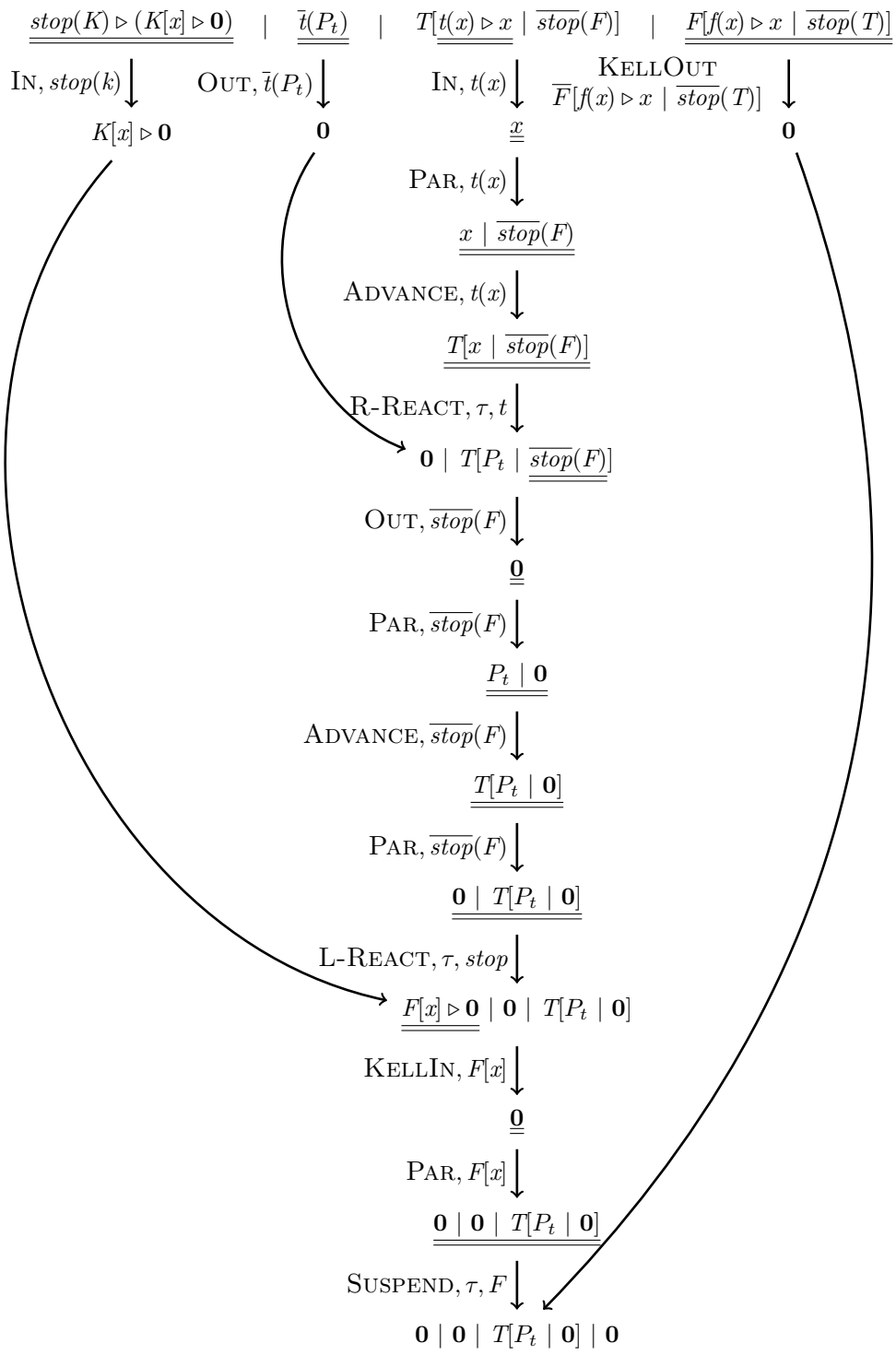


Figure 3.5: Sample Process Evolution Using LTS Semantics

$$\frac{P \xrightarrow{a(\widetilde{c})} P',\ Q \xrightarrow{\overline{a}(\widetilde{w})} Q'}{K^*[P] \mid M^*[Q] \xrightarrow{\tau} K^*[P'\{\widetilde{w}/\widetilde{c}\}] \mid M^*[Q']}\ \text{L-React}$$

$$\frac{P \xrightarrow{K[x]} P',\ Q \xrightarrow{\overline{K}[R]} Q'}{K^*[P] \mid M^*[Q] \xrightarrow{\tau} K^*[P'\{R/x\}] \mid M^*[Q']}\ \text{L-Suspend}$$

$$\frac{P \xrightarrow{a(\widetilde{d})} P',\ Q \xrightarrow{\overline{a}(\widetilde{w})} Q',\ \widetilde{c} \subseteq \widetilde{w},\ (\textbf{new}\ c\ ) \in \widetilde{w} \text{ if } c \in \widetilde{c}}{K^*[P] \mid M^*[Q] \xrightarrow{\tau} \textbf{new}\ \widetilde{c}\ (K^*[P'\{\widetilde{w}/\widetilde{c}\}] \mid M^*[Q'])}\ \text{L-Close}$$

Figure 3.6: Generalized Communication Rules for kell-m

3.4.2 Reduction Semantics

Reduction semantics are an alternative to LTS semantics. In reduction semantics, reduction rules are used to describe the operational semantics of kell-m. The reduction rules for kell-m are listed in Figure 3.7. With reduction rules the communication between processes is inferred, directly, from the syntax of the processes instead of from transitions where abstraction and concretion actions occur.

As shown in rule ReductOutKell, reactions and suspensions can occur within kells. In this rule, $\cdots$ are used to represent zero or more (bound name set, kell-m process)-pairs composed in parallel.

In the reduction rules, we assume there are no name conflicts for bound names. As we discuss at the beginning of Section 3.4 when presenting the structural equivalences for kell-m, since $K[\textbf{new}\ a\ P] \not\equiv \textbf{new}\ a\ (K[P])$, name restrictions within kells need to be treated with care. The reduction rule L-ReductExtrusion explicitly handles name extrusion from kells.

When illustrating reductions of process expressions, we sometimes write the name of the channel involved in a communication action, e.g., $P \rightarrowtail {}^{a}Q$.

R-* reduction rules can be trivially deduced by first using the ReductStruct rule, and then the corresponding L-* rule.

For example, a process P defined as:

$$P : K[\textbf{new}\ a,\ b\ (\overline{c}(\overline{a}(b)) \mid a(d) \triangleright Q)] \mid c(x) \triangleright x$$

can be reduced as follows:

$$\begin{array}{rll} P \rightarrowtail {}^{c} & \textbf{new}\ a,\ b\ (K[\textbf{0} \mid a(d) \triangleright Q] \mid \overline{a}(b)) & \text{L-ReductExtrusion} \\ \rightarrowtail {}^{a} & \textbf{new}\ a,\ b\ (K[\textbf{0} \mid Q\{b/d\}] \mid \textbf{0}) & \text{ReductOutKell} \end{array}$$

$$\frac{}{a(\widetilde{c}) \triangleright P \mid \overline{a}(\widetilde{w}) \rightarrowtail P\{\widetilde{w}/\widetilde{c}\} \mid \mathbf{0}} \text{ L-ReductReact}$$

$$\frac{}{M[x] \triangleright P \mid M[P_M] \rightarrowtail P\{P_M/x\} \mid \mathbf{0}} \text{ L-ReductSuspend}$$

$$\frac{P \rightarrowtail P'}{\mathbf{new}\ c\ P \rightarrowtail \mathbf{new}\ c\ P'} \text{ ReductRestrict}$$

$$\frac{P \rightarrowtail P'}{K[P] \rightarrowtail K[P']} \text{ ReductInKell} \qquad \frac{P \rightarrowtail P'}{P|Q \rightarrowtail P'|Q} \text{ ReductPar}$$

$$\frac{P' \equiv P,\ P \rightarrowtail Q,\ Q \equiv Q'}{P' \rightarrowtail Q'} \text{ ReductStruct}$$

$$\frac{P_d\{\widetilde{w}/\widetilde{c}\} \rightarrowtail Q,\ p(\widetilde{c}) \stackrel{def}{=} P_d}{p(\widetilde{w}) \rightarrowtail Q} \text{ ReductProc}$$

$$\frac{P \mid Q \rightarrowtail P' \mid Q'}{K^*[P \cdots] \mid M^*[Q \cdots] \rightarrowtail K^*[P' \cdots] \mid M^*[Q' \cdots]} \text{ ReductOutKell}$$

$$\frac{(\mathbf{new}\ \widetilde{c}\ P) \mid Q \rightarrowtail \mathbf{new}\ \widetilde{c}\ (P' \mid Q')}{K^*[\mathbf{new}\ \widetilde{c}\ P \cdots] \mid M^*[Q \cdots] \rightarrowtail \mathbf{new}\ \widetilde{c}\ (K^*[P' \cdots] \mid M^*[Q' \cdots])} \text{ L-ReductExtrusion}$$

Figure 3.7: Reduction Rules for kell-m

When dealing with higher-order expressions, the scope extrusion occurs for any name in the expression that is bound when the expression is output via a channel ($\overline{c}(\overline{a}(b))$ in the example).

As observed in [132], although in process algebras reduction rules are simpler than LTS semantics in the sense that there are typically less reduction rules than LTS transition rules and the reduction rules are unlabelled, there is loss of information when compared to LTS transitions. For example, consider the process $\overline{a}(w)$. This process cannot be reduced. It, nevertheless, has the potential to communicate with another process via the channel

a. Such potential for communication is manifest in the OUT LTS transition, but is lost when using reduction semantics. As we will see in Section 3.5, this lost information is somehow recovered when assuming a global observer capable of sensing, at a given time, which channels are capable of communication.

3.5 Behavioural Equivalences

We are interested in knowing if two kell-m processes exhibit the same behaviour. As proposed by Sangiorgi when dealing with higher-order calculi [144], we assume a global observer capable of sensing the communication actions that are enabled on a given channel. One can think of the global observer as another process trying to interact with the observed process.

We write $P \downarrow_{\overline{a}}$ (similarly, $P \downarrow_a$) to specify that a concretion (similarly, abstraction) is enabled on channel a. $P \downarrow_{\overline{a}}$ and $P \downarrow_a$ are called *visibility predicates*, and we frequently write $P \downarrow_\alpha$ to indicate an arbitrary visibility predicate.

The visibility predicates for kell-m are determined by the function $\mathcal{V}$, defined in Figure 3.8. According to $\mathcal{V}$'s definition, $P \downarrow_\alpha$, if $\alpha \in \mathcal{V}[\![P]\!]$. Hence, $a \in \mathcal{V}[\![P]\!]$ if exists P', such that $P \xrightarrow{a(\widetilde{c})} P'$ or $P \xrightarrow{a[x]} P'$. Both, $a(\widetilde{c})$ and $a[x]$, are abstractions representing patterns in trigger expressions. $a(\widetilde{c})$ matches a write on channel a, and $a[x]$ matches a kell named a. Similarly, $\overline{a} \in \mathcal{V}[\![P]\!]$ if $P \xrightarrow{\overline{a}(\widetilde{w})} P'$ or $P \xrightarrow{\overline{a}[Q]} P'$. Both, $\overline{a}(\widetilde{w})$ and $\overline{a}[Q]$ are concretions. $\overline{a}(\widetilde{w})$ represents a write on channel a, and $\overline{a}[Q]$ represents a kell $a[Q]$.

Notice restricted names (names in **new** ... expressions) are not visible. The reason is that, as previously mentioned, one can think of the global observer as another process trying to interact with the observed process. Restricted names are not available to the global observer process, and therefore actions on these channels are not visible to the global observer.

Also notice that τ is not included as a visibility predicate. The reason is that τ actions in the observed process are actions internal to the process and cannot be sensed by the global observer.

3.5.1 Barbed Bisimulation

Using P^k to represent the class of kell-m processes, and based on Sangiorgi's behavioural equivalences for higher-order process calculi [144], a *barbed simulation* for the kell-m calculus is any relation S such that $S \subseteq P^k \times P^k$, and $(P, Q) \in S$ if:

$$
\begin{array}{lcl}
\mathcal{V}[\![\ \mathbf{0}\ ]\!] & \stackrel{def}{=} & \{\} \\
\mathcal{V}[\![\ x\ ]\!] & \stackrel{def}{=} & \{\} \\
\mathcal{V}[\![\ \mathbf{new}\ a\ P\ ]\!] & \stackrel{def}{=} & \mathcal{V}[\![\ P\ ]\!] \setminus \{a\} \\
\mathcal{V}[\![\ \overline{a}(\widetilde{w})\ ]\!] & \stackrel{def}{=} & \{\overline{a}\} \\
\mathcal{V}[\![\ K[P]\ ]\!] & \stackrel{def}{=} & \{\overline{K}\} \cup \mathcal{V}[\![\ P\ ]\!] \\
\mathcal{V}[\![\ a(\widetilde{c}) \triangleright P\ ]\!] & \stackrel{def}{=} & \{a\} \\
\mathcal{V}[\![\ K[x] \triangleright P\ ]\!] & \stackrel{def}{=} & \{K\} \\
\mathcal{V}[\![\ P|Q\ ]\!] & \stackrel{def}{=} & \mathcal{V}[\![\ P\ ]\!] \cup \mathcal{V}[\![\ Q\ ]\!]
\end{array}
$$

Figure 3.8: Visibility Predicates for kell-m

1. $P \xrightarrow{\tau} P'$, then $\exists Q' : Q \xrightarrow{\tau} Q'$, and $(P', Q') \in S$, and

2. If $P \downarrow_\alpha$, then $Q \downarrow_\alpha$

Alternatively, using the reduction rules instead of the LTS semantics, condition 1 in the definition of barbed simulation relations can be replaced by:

1. $P \rightarrowtail P'$, then $\exists Q' : Q \rightarrowtail Q'$ $(P', Q') \in S$, and

condition 2 remains unaltered.

The reason why barbed simulation can be defined with the communication transitions $\xrightarrow{\tau}$ or using the reduction relation, is that the reduction relations represent, the $\xrightarrow{\tau}$ transitions of the labelled transition semantics, except for the differences accounted by structural equivalences [117].

A relation S is a *barbed bisimulation*, if both S and S^{-1} are barbed simulations. *Weak* definitions of the barbed simulation and bisimulations are obtained by replacing $\rightarrow$ with $\Rightarrow$ in the definitions, where $\Rightarrow$ is the reflexive and transitive closure of $\rightarrow$ (or $\rightarrowtail$ if using the reduction relation).

A process P is said to be *barbed similar* to a process Q if there is a barbed simulation S such that $(P, Q) \in S$. Similarly, P and Q are *barbed bisimilar* if there is a barbed bisimulation S such that $(P, Q) \in S$.

For example, consider processes P and Q defined as:

$$
\begin{array}{ll}
P: & \overline{a}(\overline{b}() \mid b() \triangleright \mathbf{0}) \mid a(x) \triangleright x \\
Q: & \mathbf{new}\ h\ (h() \triangleright (\overline{b}() \mid b() \triangleright \mathbf{0}) \mid \overline{a}(h) \mid a(c) \triangleright \overline{c}())
\end{array}
$$

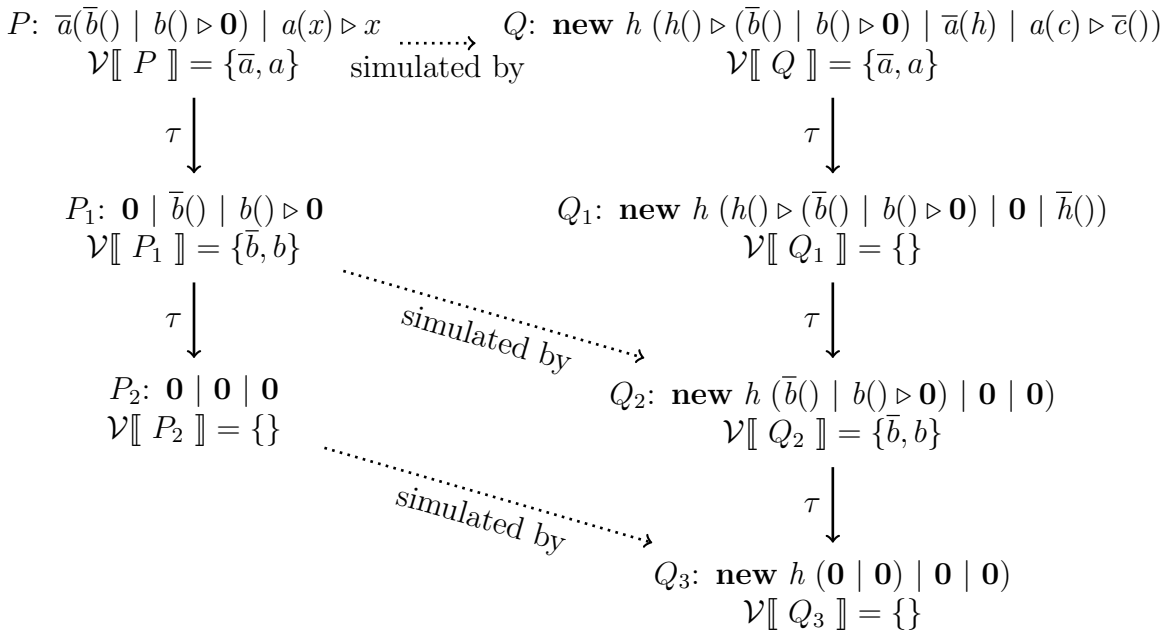


Figure 3.9: Example of Weak Barbed Simulation in kell-m

Intuitively, Q and P have the same behaviour: the potential for communication on a is visible in both process, and later, the potential for communication on b is visible. The communication transitions and visibility predicates for each one of the processes are illustrated in Figure 3.9.

Notice in Figure 3.9 Q is not barbed similar to P. The reason is the transition from Q_1 to Q_2 which is not matched in P. A weak barbed simulation S for P and Q is $\{(P, Q), (P_1, Q_2), (P_2, Q_3)\}$. S is represented in the figure by the dotted arrows.

Even when two processes are barbed bisimilar, their behaviour may not be the same when used along other processes. For example, consider the following processes:

$$\begin{aligned} P: &\quad \overline{a}(c) \mid \overline{b}(u) \mid b(l) \triangleright \mathbf{0} \\ Q: &\quad \overline{a}(d) \mid \overline{b}(u) \mid b(w) \triangleright \mathbf{0} \end{aligned}$$

A barbed bisimulation for them is:

$$S = \{(P, Q), (\overline{a}(c) \mid \mathbf{0} \mid \mathbf{0}, \overline{a}(d) \mid \mathbf{0} \mid \mathbf{0})\}$$

But when used along the process $R : a(e) \triangleright \overline{e}()$, the visibility predicates of $P|R$ and $Q|R$ are different. After P and R interact, the visibility predicate is $\{\overline{c}, \overline{b}, b\}$; if Q and R interact, the visibility predicate is $\{\overline{d}, \overline{b}, b\}$.

To define behavioural similarity under any use, it is necessary to introduce contexts $\mathcal{C}$. Assuming arbitrary kell-m processes R and P, P can be used in the following contexts $\mathcal{C}$:

$$\mathcal{C} \quad ::= \quad P \;\mid\; \textbf{new}\ a\ P \;\mid\; \overline{a}(P) \;\mid\; P|R \;\mid\; R|P \;\mid\; a(\widetilde{c}) \triangleright \mathcal{C} \;\mid\; K[\mathcal{C}] \;\mid\; K[x] \triangleright \mathcal{C}$$

Two processes P and Q are *barbed congruent* if they are barbed bisimilar under all contexts. When two processes are barbed congruent, they can be used interchangeably. Showing that two processes are barbed congruent requires showing that there is no context under which the processes behave differently. A differentiating context may be easy to find for some processes that are not congruent, but proving that such a context does not exist for processes suspected congruent can be a difficult task [146]. Although we were unable to find in the literature an actual result for the complexity of this task, Davide Sangiorgi has this to say [146]:

> Context-based behavioural equalities like barbed congruence suffer from the universal quantification on contexts, that makes it very hard to prove process equalities following the definition, and makes mechanical checking impossible.

We conclude the presentation of barbed bisimulation in kell-m by demonstrating that barbed congruent processes have the same kell structure. Recall the kell structure of a process determines the containment relationships between kells in the process. By contradiction, let us assume processes P and Q are barbed congruent but they do not have the same kell structure. If P and Q do not have the same kell structure, it is because one of the following cases:

- There is at least a kell K in one processes and not in the other. Without loss of generality, let us assume K is in P and not in Q. A differentiating context is $C_1 : K[x] \triangleright \mathbf{0}$. This is because $P|C$ has at least one transition labelled τ due to the passivation of K. That transition does not occur in $Q|C$. Moreover, K is in the visibility predicate for $P|C$ but not for $Q|C$.

- A kell K is located in another kell L in one of the processes but not in the other. Both processes P and Q have kells K and L, otherwise the previous case applies. Without loss of generality, let us assume kell K is located in kell L in P but not in Q. A differentiating context in this case is $C_2 : L[x] \triangleright K[y] \triangleright \mathbf{0}$. With this context, $Q|C$ has an extra transition labelled τ, corresponding to the passivation of K, that does not occur in $P|C$. The reason is that in $P|Q$, because K is located in L, the passivation of K cannot occur once L is passivated.

In both cases we were able to exhibit differentiating contexts. Hence, if P and Q are barbed congruent, then they have the same kell structure.

Besides C_1 and C_2 above, other differentiating contexts may exist for particular processes P and Q. For example, consider:

$$P : K[a(c) \triangleright \mathbf{0}] \mid T[\mathbf{0}] \mid \overline{a}(w)$$
$$Q : K[\,T[a(c) \triangleright \mathbf{0}]] \mid \overline{a}(w)$$

Processes P and Q are barbed bisimilar: a barbed bisimulation for them is:

$$S = \{(P, Q), (K[\mathbf{0}] \mid T[\mathbf{0}] \mid \mathbf{0}, K[\,T[\mathbf{0}]] \mid \mathbf{0})\}$$

A differentiating context is $T[x] \triangleright \mathbf{0}$. $P|C$ produces $K[a(c) \triangleright 0] \mid \mathbf{0} \mid \overline{a}(w)$, while $Q|C$ produces $K[\mathbf{0}] \mid \overline{a}(w)$. These resulting processes are not barbed bisimilar.

3.5.2 Bisimulation up to Kell Containment

We are interested in verifying if two processes have the same kell structure. As discussed at the end of the previous section, an avenue is to check if they are barbed congruent. Although, because of its difficulty, relying on barbed congruence should be avoided. Moreover, barbed congruency is a sufficient, but not a required, condition for kell structure similarity. For example, consider:

$$Q : K[\,T[a(c) \triangleright \mathbf{0}]] \mid \overline{a}(w)$$

Processes $Q|\overline{b}(e)$ and $Q|\overline{b}(l)$ are not barbed congruent. A differentiating context in this case is C: $b(r) \triangleright \overline{r}()$. They have, nevertheless, the same kell structure.

Using P^k to represent the class of kell-m processes, a *simulation up to kell containment* for kell-m is any relation S such that, $S \subseteq P^k \times P^k$, and $(P, Q) \in S$ if:

$$P \xrightarrow{\overline{K}[.]} P', \text{ then } \exists Q' : Q \xrightarrow{\overline{K}[.]} Q', \text{ and } (P', Q') \in S$$

The dot is used to indicate that we are not interested in the contents of the concretion.

A relation S is a *bisimulation up to kell containment*, if both S and S^{-1} are barbed simulations up to kell containment. As with the regular barbed bisimulations, *weak* definitions of the barbed simulation and bisimulations up to kell containment are obtained by replacing $\rightarrow$ with $\Rightarrow$ in the definitions, where $\Rightarrow$ is the reflexive and transitive closure of $\rightarrow$.

Two processes have the same structure if and only if they are bisimilar up to kell containment. We first show that if processes P and Q are bisimilar up to kell containment, then they have the same kell structure. We argue by contradiction. Let us assume that P and Q are bisimilar up to kell containment but they do not have the same kell structure. The possible situations under which P and Q do not have the same kell structure are:

- There is a kell K in one of the processes but not in the other. Without loss of generality, let us assume K is in P but not in Q. Because of rule KellOut (cf. Figure 3.4), a possible transition for P is $P \xrightarrow{\overline{K}[R]} P'$, where R is the process located in K. Since K is not in Q such transition does not exist in Q and, by definition, P and Q cannot be bisimilar up to kell containment.

- A kell K is located within a kell T in one of the processes but not in the other. Without loss of generality we assume kell K is located in kell T in P but not in Q. Both kells, K and T, are in Q, otherwise the previous case applies. Because of rule KellOut, P can transition to $P \xrightarrow{\overline{T}[U]} P'$ where R is the process located in T. By the definition of bisimilarity up to kell containment, Q can also transition to Q': $Q \xrightarrow{\overline{T}[U]} Q'$. Since T has been passivated, and because of our assumption about the location of K in P, K is not in P'. Because our assumption of the location of K in Q, K has not been passivated in Q', and Q' can transition to $Q' \xrightarrow{\overline{K}[U']} Q''$, which is a transition that cannot be matched from P'. Therefore, by definition, P and Q cannot be bisimilar up to kell containment.

We now show that if processes P and Q have the same kell structure, then they are bisimilar up to kell containment. Again, by contradiction, let us assume that P and Q have the same kell structure but they are not bisimilar up to kell containment. If P and Q are not bisimilar up to kell containment this is because one of the following reasons:

- There is a transition from $P \xrightarrow{\overline{K}[R]} P'$ that is not matched from Q. This means a kell K is in P and not in Q. Therefore P and Q do not have the same kell structure.

- There is a transition from $Q \xrightarrow{\overline{K}[R]} Q'$ that is not matched from P. The argument is the same as the previous case.

- P and Q have matching transitions up to processes P'' and Q'' at which point there is a transition from P'' that cannot be matched from Q''. That transition represents a kell that is in P and not in Q. Therefore P and Q do not have the same kell structure.

- P and Q have matching transitions up to processes P'' and Q'' at which point there is a transition from Q'' that cannot be matched from P''. The argument is the same as the previous case.

Hence, to verify if two processes have the same kell structure, a bisimulation up to kell containment has to be produced.

3.5.3 Extended Semantics

As we will see in Chapter 4, it is useful not only to check if two processes have the same kell structure, but also to have the ability to verify kell containment conditions. For example, one may be interested in checking if a given action occurs, or does not occur, within a given kell. Based on the operational semantics for kell-m we have presented so far, there is no simple way to check such kell containment conditions.

Recall the LTS semantics for kell-m, presented in Section 3.4.1 and listed in Figure 3.4. We now extend kell-m's LTS operational semantics with kell containment information, and expose the concretion and abstraction actions matched in τ communications. The rules for the extended LTS semantics are listed in Figure 3.10. Specifically, the changes are:

- For abstraction and concretion actions, a set κ is included with the names of kells where the action is located.
- τ transitions are decorated with the name of the channel or kell involved in the communication, the parameters of the communication, and the kell containment sets for the matched abstraction and concretion.
- To further differentiate τ actions from abstraction and concretions, channel and kell names in the communication are decorated with an over double arrow, e.g., $\overleftrightarrow{a}(\widetilde{w})$, $\overleftrightarrow{K}[P]$.
- α in the transitions represents abstractions (i.e., $a(\widetilde{c})$, $K[x]$), concretions (i.e., $\overline{a}(\widetilde{w})$, $K[P]$). α_τ represents τ actions (i.e., $\overleftrightarrow{a}(\widetilde{w})$, $\overleftrightarrow{K}[P]$)
- δ is used to represent any kind of transition. Therefore, δ depicts a triple $(\alpha_\tau, \kappa_a, \kappa_c)$ for τ actions, and a pair (α, κ) for abstractions and concretions.
- To differentiate the new rules from the rules in Section 3.4.1, we prefix the new rules with the letter X.

We will assume XR-*, similar to the communication rules XL-REACT, XL-SUSPEND, and XL-CLOSE, but with the order of the reader and writer processes inverted. The XR-* are obtained by applying the XSTRUCT rule, and then the corresponding XL-* rule.

XRESTRICT removes restricted names from the kell containment set. When a process is located within a kell, if the process can advance with concretion or abstraction actions, rule XADVANCE adds the kell to the kell containment set.

Using extended semantics, Figure 3.11 shows the possible evolution of a processes $Q : K[\,T[a(c) \triangleright \mathbf{0}]\,] \mid L[\overline{a}(w)]$.

$$\frac{P \xrightarrow{\delta} R,\ P \equiv Q}{Q \xrightarrow{\delta} R}\ \text{XStruct}$$

$$\overline{a}(\widetilde{w}) \xrightarrow{\overline{a}(\widetilde{w}),\{\}} \mathbf{0}\quad \text{XOut} \qquad\qquad a(\widetilde{c}) \triangleright P \xrightarrow{a(\widetilde{c}),\{\}} P\quad \text{XIn}$$

$$K[P] \xrightarrow{\overline{K}[P],\{\}} \mathbf{0}\quad \text{XKellOut} \qquad\qquad K[x] \triangleright P \xrightarrow{K[x],\{\}} P\quad \text{XKellIn}$$

$$\frac{P \xrightarrow{\alpha,\kappa} Q,\ c \notin bn(\alpha)}{\mathbf{new}\ c\ P \xrightarrow{\alpha,\kappa\setminus\{c\}} \mathbf{new}\ c\ Q}\ \text{XRestrict}$$

$$\frac{P \xrightarrow{\alpha_\tau,\kappa_a,\kappa_c} Q}{\mathbf{new}\ c\ P \xrightarrow{\alpha_\tau,\kappa_a\setminus\{c\},\kappa_c\setminus\{c\})} \mathbf{new}\ c\ Q}\ \text{XRestrictTau}$$

$$\frac{P \xrightarrow{\alpha,\kappa} Q,\ K \notin bn(\alpha)}{K[P] \xrightarrow{\alpha,\kappa\cup\{K\}} K[Q]}\ \text{XAdvance}$$

$$\frac{P \xrightarrow{\alpha_\tau,\kappa_a,\kappa_c} Q}{K[P] \xrightarrow{\alpha_\tau,\kappa_a\cup\{K\},\kappa_c\cup\{K\}} K[Q]}\ \text{XAdvanceTau}$$

$$\frac{P \xrightarrow{\delta} Q, (\delta = (\alpha,\kappa)) \Rightarrow (bn(\alpha) \cap fn(R) = \emptyset)}{P|R \xrightarrow{\delta} Q|R}\ \text{XPar}$$

$$\frac{P_d\{\widetilde{w}/\widetilde{x}\} \xrightarrow{\delta} Q,\ p(\widetilde{x}) \stackrel{def}{=} P_d}{p(\widetilde{w}) \xrightarrow{\delta} Q}\ \text{XProc}$$

$$\frac{P \xrightarrow{\overline{a}(\widetilde{w}),\kappa} Q,\ c \in names(\widetilde{w}),\ c \neq a}{\mathbf{new}\ c\ P \xrightarrow{\overline{a}(\widetilde{w}'),\kappa\setminus\{c\}} Q,\ \text{with } \widetilde{w}' = \widetilde{w}\{\mathbf{new}\ c\ /c\}}\ \text{XOpen}$$

$$\frac{P \xrightarrow{a(\widetilde{c}),\kappa_a} P',\ Q \xrightarrow{\overline{a}(\widetilde{w}),\kappa_c} Q'}{P \mid Q \xrightarrow{\overleftrightarrow{a}(\widetilde{w}),\kappa_a,\kappa_c} P'\{\widetilde{w}/\widetilde{c}\} \mid Q'}\ \text{XL-React}$$

$$\frac{P \xrightarrow{K[x],\kappa_a} P',\ Q \xrightarrow{\overline{K}[R],\kappa_c} Q'}{P \mid Q \xrightarrow{\overleftrightarrow{K}[R],\kappa_a,\kappa_c)} P'\{R/x\} \mid Q'}\ \text{XL-Suspend}$$

$$\frac{P \xrightarrow{a(\widetilde{d}),\kappa_a} P',\ Q \xrightarrow{\overline{a}(\widetilde{w}),\kappa_c} Q',\ \widetilde{c} \subseteq \widetilde{w},\ (\mathbf{new}\ c\ ) \in \widetilde{w} \text{ if } c \in \widetilde{c}}{P \mid Q \xrightarrow{\overleftrightarrow{a}(\widetilde{w}),\kappa_a,\kappa_c} \mathbf{new}\ \widetilde{c}\ (P'\{\widetilde{w}/\widetilde{c}\} \mid Q')}\ \text{XL-Close}$$

Figure 3.10: Kell-m Extended Labelled Transition System Semantics

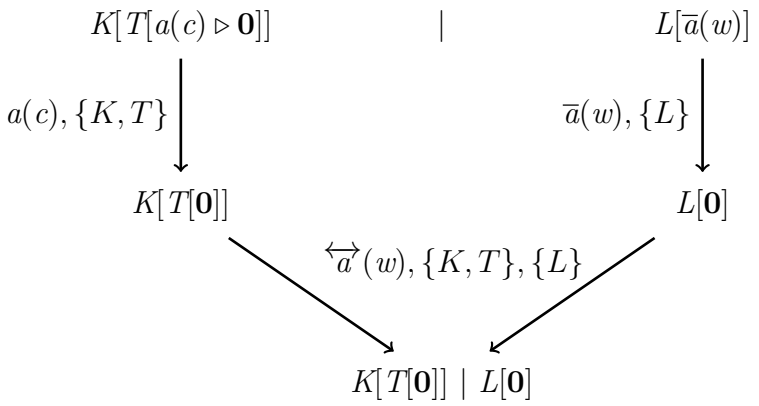


Figure 3.11: Example of Process Evolution with Extended LTS Semantics

The extended LTS semantics are the basis of kμ, our logic for the specification of properties in systems modelled using kell-m (cf. Chapter 4). When τ actions expose the information of the abstraction and concretion actions being matched, they are no longer invisible to an observer. With kμ, this allows the specification of properties that can impose conditions at the time processes communicate. For example, one may be interested in specifying a condition such as *process in kell K_1 communicates with process in kell K_2 using channel a.*

Also, as shown in Chapter 6, using extended semantics it is possible to encode both the LTS and reduction semantics into a single tool. In reduction semantics, only τ transitions are considered. In LTS semantics, abstraction and concretion transitions are considered, and τ transitions are not inspected for matching abstraction and concretion information.

This concludes our formal presentation of kell-m. In the next section we compare kell-m with related process algebras.

3.6 Related Work

Process algebras were conceived for the study of concurrent and parallel systems at the end of the 1970s [114, 13, 18]. Based on one of these algebras, the Calculus of Communicating Systems [114], Robin Milner, Joachim Parrow, and David Walker developed the π-calculus [116]. Process expression in the π-calculus have the following syntax:

$$P ::= \mathbf{0} \;\mid\; \overline{a}(\widetilde{c}).P \;\mid\; a(\widetilde{c}).P \;\mid\; P + P \;\mid\; P|P \;\mid\; \mathbf{new}\ a\ P \;\mid\; P!$$

$\mathbf{0}$ represents the null process. $\overline{a}(\widetilde{c}).P$ represents a process writing names $\widetilde{c}$ on channel a. After the names are read, the process continues as P. $a(\widetilde{c}).P$ represents a process waiting

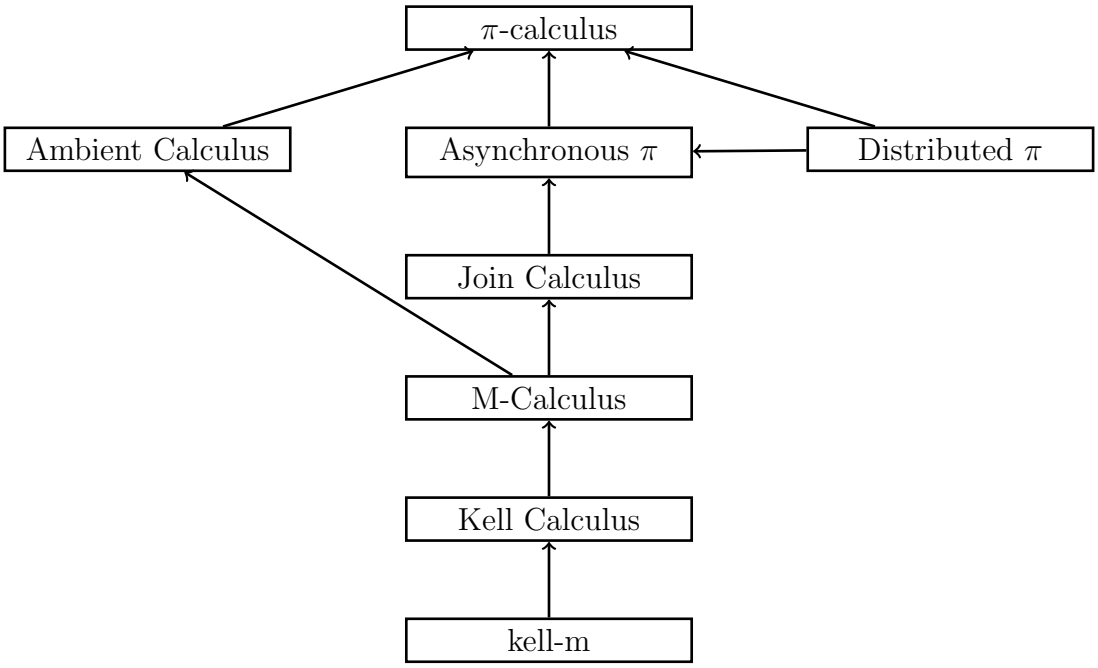


Figure 3.12: Lineage of kell-m

for input on channel a. When the input is received, the process continues as P. $P + Q$, called *summation* of processes, represents a process that behaves either like P or Q. $!P$, called *replication*, represents an infinite number of copies of P, running in parallel. Some variations of the π-calculus support recursive processes in-lieu of replication. Because π-calculus is a first-order calculus, only names can be transmitted in a communication.

The asynchronous π-calculus was one of the first variations proposed of the π-calculus [87, 25]. Like kell-m, in the asynchronous π-calculus outputs on channels cannot be followed by a process, and the summation of processes is eliminated from the algebra. Many other process algebras proposed for studying distributed systems, including kell-m, can trace their lineage back to the π-calculus (cf. Figure 3.12).

In this section we present several process algebras closely related to kell-m. Most of these algebras provide some explicit notion of locality, that can be used to represent physical locations in a distributed system or, in the case of kell-m, to model control and modularization constructs. The algebras considered are distributed π, ambient calculus, join calculus, M-calculus, and Kell calculus.

Since key to our development of kell-m is the ability to verify if given properties are met by a system represented using kell-m, we note that with the exception of π-calculus [160, 171, 65], and the ambient calculus [35, 147, 41], we are not aware of model checking

support for any of the other presented algebras. Actual comparison of our model checking approach with that of other model checkers based on process algebras can be found in Section 6.5.

3.6.1 Distributed π

The distributed π calculus, Dπ, is a first-order calculus. There are both, synchronous and asynchronous versions of Dπ [82, 80]. Dπ was the first variation of the π-calculus to introduce an explicit notion of locality. Associated to each location is one or more processes (called *threads*) running in parallel. An expression $K[P] \mid K[R]$, specifies that processes R and P are executing both in location K. Notice, in kell-m, such an expression would represent instead two locations named K, each executing a different process. Locations in the Dπ are not themselves processes, therefore locations cannot be located within other locations.

When two processes at different locations wish to communicate, one of the processes must migrate to the location of the other process. For this purpose, Dπ provides a *move* operator $T :: P$ that allows the migration of process P to location T. For example, in $K[T :: P] \mid T[Q]$ process P migrates to location T resulting in $T[P] \mid T[Q]$. When a process migrates to another location it looses the communication channels of the previous location.

Locations in Dπ cannot be passivated. Hence the only way to alter the process executing at a location is by shipping code to the location via the move operator. There is no way to stop or terminate a location.

3.6.2 Ambient Calculus

In the ambient calculus processes execute within bounded places called *ambients* [34]. Ambients themselves are processes and can be contained within other ambients. In contrast to the π-calculus and other similar process algebras (including kell-m), the basic abstraction for interaction in the original ambient calculus is not channel communication, but the movement of ambients from one ambient to another. The following is the syntax for processes in the ambient calculus:

$$P ::= \mathbf{0} \mid P|P \mid \mathbf{new}\ a\ P \mid P! \mid K[P] \mid \mathbf{in}\ K.P \mid \mathbf{out}\ K.P \mid \mathbf{open}\ K.P$$

in $K.P$ instructs the ambient surrounding **in** $K.P$ to enter the sibling ambient K:

$$T[\mathbf{in}\ K.P|Q] \mid K[R] \rightarrow K[R \mid T[P|Q]]$$

out $K.P$ instructs the ambient surrounding **out** $K.P$ to exit its parent ambient K:

$$K[T[\textbf{out } K.P|Q] \mid R] \rightarrow K[R] \mid T[P|Q]$$

If the parent ambient is not named K, the operation is blocked until K becomes its parent.

open $K.P$ dissolves the ambient K located at the same level as **open** $K.P$:

$$\textbf{open } K.P \mid K[R] \rightarrow P \mid R$$

If there is no ambient K at the same level, the operation is blocked until one exists.

As previously mentioned, there are no channel communications in the ambient calculus. Extensions exist that allow local communication within ambients, [e.g., 28, 96]. In boxed ambients [28], the **open** construction is no longer used, and communication is done via anonymous (unnamed) channels: $(x)^K$ is used to represent input x from ambient K; $\overline{x}^{\uparrow}$ is used to represent output for the parent ambient.

In contrast with kell-m, ambients (locations) in the ambient calculus can only communicate with neighbouring ambients. Ambient passivation is not supported, in the sense that the process running inside an ambient cannot be destroyed. Also, although named channels can be simulated using anonymous channels in the boxed ambient calculus, care needs to be taken to avoid unintended communications. This is because, effectively, the form of channel communication in boxed ambients can be seen as having only one named channel, shared by all neighbouring processes.

Although extensive research has been done related to model checking of systems represented using the ambient calculus ([e.g., 35, 85, 40, 41, 106]), we are not aware of any model checking tool support. The ambient logic, a modal logic for the ambient calculus is proposed by Cardelli and Gordon in [35], along with a model checker algorithm that receives as input a system represented as an ambient calculus expression, and a property expressed in ambient logic, and verifies if the given system satisfies the given property. The verification problem for the ambient calculus is undecidable when the calculus has replication or certain operators are used in the ambient logic [41]. In contrast, verification for kell-m is decidable if the LTS for the process being verified is finite.

3.6.3 Join Calculus

In the join calculus with locations [70], abstractions are represented as patterns in trigger expressions similar to those of kell-m. In contrast to kell-m, the patterns can specify messages coming on multiple channels, for example:

$$(a(\widetilde{c}) \mid b(\widetilde{e})) \triangleright P$$

These patterns are called *multi-join* patterns and are matched only after write operations have occurred in all the channels specified in the pattern. Both, synchronous and asynchronous channels are supported in the join calculus. When synchronous channels are used, the trigger reacting to a write must return a value back to the writer.

JoCaml [6], is a partial implementation of the join calculus using OCaml [7]. Patterns in JoCaml can specify conditions on the values received on a channel [102]. In this case, the pattern is only matched if there are writes on the channels specified in the pattern, and the values written on the channels match the conditions specified by the pattern. Besides values, OCaml functions can be passed as part of communications. A function received on a channel can then be run as a process. Hence, JoCaml is a higher-order implementation of the join calculus.

Processes in the join calculus can be located on different sites, and sites can be within other sites. Site migration is supported via a $\mathbf{go}(K, a)$ operator, where K is the new location of the site within which the **go** operator is currently executing and a is a channel. Once the location is moved to its new location, a write on channel a is performed. Hence, other process can wait for a site to be moved. In the following example, site T is moved to site K, and a process on site L is notified when the move has occurred:

$$T[P \mid \mathbf{go}(K, n)] \mid U[K[Q]] \mid L[n() \triangleright R] \Rightarrow U[K[Q \mid T[P]]] \mid L[R]$$

In JoCaml sites represent actual computers and cannot be located within other sites. As with kell-m, both in the join calculus and JoCaml, two processes can communicate via a channel, independently of their locations. Similar to kell-m and its encoding (cf. Chapter 6), when a function in JoCaml (a process in the case of kell-m) is sent to a remote process, the names local to the function are also sent along with the process. In contrast to kell-m, JoCaml does not support passivation of locations nor, as previously mentioned, does it allow the migration of one location to another location.

3.6.4 M-Calculus

A precursor of the Kell calculus, the M-calculus is an attempt at defining a formal distributed programming model [149]. M-calculus is a synchronous, higher-order calculus, derived from the join calculus. Besides processes composed from channel communications, processes in the M-calculus can also be functions represented as lambda abstractions $\lambda x.P$. Sites in M-calculus are called *cells*, and cells are themselves processes, allowing sites to be structured hierarchically. A cell ϵ is the root of the site hierarchy and is not included within any other site. Every process, with the exception of cell ϵ, must be located within a site.

A process $K(P)[Q]$ represents a cell K with *membrane* process P, and *plasma* process Q. The membrane process filters incoming and outgoing messages. The plasma process is the actual content of the cell.

The M-calculus is the first calculus to introduce a passivation operator, but its semantics are different than those of kell-m and Kell calculus. In the M-calculus passivation of a cell can only be done by the cell's own membrane processes, effectively suspending the membrane and plasma processes of the cell. The passivation operator, named **pass**, has as its only argument the a name of a function. When a cell is passivated by its membrane, the name of the cell as well as its membrane and plasma processes are passed to the function specified when **pass** was invoked.

Similarly to the join calculus, channel abstractions in the M-calculus are expressed as multi-join triggered patterns. In contrast to kell-m, when the pattern of a trigger is matched the trigger does not disappear. Hence, all triggers in the M-calculus are recurrent (cf. Section 3.2 for the definition of recurrent triggers).

When a process writes on a channel, the membrane process of the cell where the process is contained must decide whether to allow the message out or not. Two special names i and o are accessible in the M-calculus to every membrane process. In i the process receives incoming messages and in o it receives outgoing messages. For each message, incoming or outgoing, three values are received: the name of the destination site, the name of the destination channel, and the actual values sent in the communication. A match operator, $[a = b]$, allows membrane processes to compare names and decide whether to let a message in or out. For example, the following expression in a membrane process:

$$i(d,\ r,\ v) \triangleright [d = a]\overline{b.r}(v)$$

specifies that messages not destined to site a are discarded. Any message destined to a is sent to channel r, at site b, instead.

The ability to control the routing of messages makes the calculus rather complex, as the creators of the calculus observe in [150], and the development of a bisimulation theory for the calculus has never been attempted.

3.6.5 Kell Calculus

At least three different, in some cases incompatible, operational semantics have been proposed for the Kell calculus [155, 150, 21]. For notational convenience, we refer to the calculus defined in [155] as *Kell1*, the calculus defined in [150] as *Kell2*, and the calculus defined in [21] as *Kell3*.

Channel communications in kell-m, Kell1 and Kell3 are asynchronous, and synchronous in Kell2. $\overline{a}(c).P$ is a valid process in Kell2, meaning that, after $\overline{a}(c)$ is matched to a corresponding trigger, the process continues as P. Akin to the differences between π-calculus and asynchronous π-calculus [129], our expectation is that Kell2 is not as suitable

as kell-m, and the other Kell calculi for modelling asynchronous interactions. Although this expectation could be confirmed with a theoretical analysis, we focus instead on other differences between kell-m and the Kell calculi. These other differences are centred in the rules that govern process communication. Specifically, the way in which processes can communicate via channels, the patterns allowed in triggers, and kell passivation rules.

Channel Communication Figure 3.13 illustrates the channel communication rules in (a) Kell1 and Kell2, (b) Kell3, and (c) kell-m. Thick lines represent kell containment relationships, arrows with thin lines represent possible channel communications. In kell-m, communication can happen between any two processes independent of their kell location. This is in contrast with Kell1, Kell2, and Kell3, where channel communication between processes is restricted by the location of the process. In the case of Kell1 and Kell2 (cf. Figure 3.13 (a)), channel communication can happen only between processes separated by no more than one kell boundary. In Kell3 (cf. Figure 3.13 (c)), channel communication between processes can happen only if the processes are in the same branch of the kell containment tree.

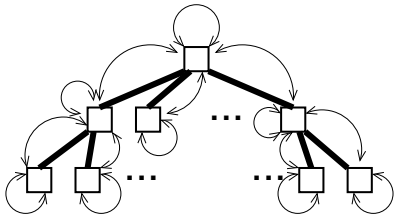
The main reason for limiting channel communication in Kell calculi is the assumption that kells represent physical hierarchical locations. Kell calculi obey a locality principle that states that any computing action should only involve one locality at a time [155, p. 7]. In kell-m, kells do not necessarily represent physical locations. Kells in kell-m are a construct that allows the association of a process with a name, as well as the passivation of the process.

In Kell calculi channel reads in trigger patterns are decorated with up and down arrows to indicate the direction, in the kell containment tree, from which the write must come. In Kell1 and Kell2, $a^{\uparrow}(c) \triangleright P$ can only be matched with a write process one level up in the kell containment tree. Similarly, $a^{\downarrow}(c) \triangleright P$ can only be matched with a write process one level down in the kell containment tree. An undecorated channel name, $a(c) \triangleright P$, can only be matched with a write occurring locally, at the same level in the kell containment tree. For example, in process:

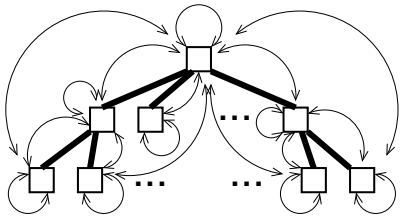
$$\overline{a}(w) \mid K[a^{\uparrow}(c) \triangleright \overline{b}(e)] \mid \ \mid b(j) \triangleright Q \mid b^{\downarrow}(h) \triangleright \mathbf{0}$$

$a^{\uparrow}(c)$ is matched with $\overline{a}(w)$, and once the communication occurs, $b^{\downarrow}(h)$ is matched with $\overline{b}(e)$. $b(j) \triangleright Q$ cannot be matched with $\overline{b}(e)$ because they are not at the same containment level, as required by the undecorated $b(j)$.

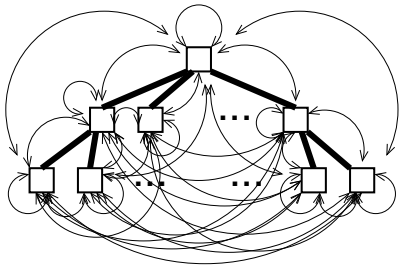
In Kell3, the meaning of the arrow decorations is different. An up (similarly, down) arrow is matched only with processes up (similarly, down) in the kell containment tree. Arrows can be decorated with a kell name, $a^{\uparrow^K}(c)$, to indicate that the message on channel a must come from kell K, located somewhere upwards (or downwards for $a^{\downarrow^K}(c)$) in the



(a) Kell1 and Kell2



(b) Kell3



(c) kell-m

Figure 3.13: Channel Communications in kell-m and Kell Calculi

kell containment tree. The meaning of an undecorated channel in a pattern expression is unchanged from Kell1 and Kell2.

Kell Passivation All three Kell calculi only allow the passivation of a kell if no kell boundaries are crossed. For example, in $K[P] \mid K[x] \triangleright Q$ the kell is passivated, but in $K[P] \mid T[K[x] \triangleright Q]$ and $T[K[P]] \mid K[x] \triangleright Q$ it is not. This is in contrast with kell-m where a kell can be passivated independently of its location.

Trigger Patterns Kell1 and Kell2 support triggers with patterns matching more than one channel or kell. For example, a valid trigger in Kell1 and Kell2 is:

$$(a(b) \mid K[x]) \triangleright P$$

Such a trigger is only matched if there is a write $\overline{a}(w)$ at the same kell containment level of the trigger and, at the same time, there is an active kell $K[Q]$. Because of the locality principle in Kell calculi, in a complex pattern all matching actions need to be in the same location. This type of trigger is not possible in Kell3 and kell-m.

Since complex triggers are not supported in kell-m, a kell-m approximation for the trigger in the example would be the process:

$$a(b) \triangleright K[x] \triangleright p$$

Nevertheless, there are situations under which this process does not have the same behaviour as the complex trigger. The reason is that the kell-m trigger does not require a write on channel a and a kell K to be active *at the same time*, as the complex trigger requires. Therefore, the following kell-m process:

$$\overline{a}(w) \mid a(b) \triangleright K[x] \triangleright P$$

evolves to:

$$\mathbf{0} \mid K[x] \triangleright P$$

while, in Kell1 and Kell2, the same process with a complex trigger cannot evolve:

$$\overline{a}(w) \mid (a(b) \mid K[x]) \triangleright P$$

This is a shortcoming of kell-m when compared with Kell1 and Kell2. However, the need for complex triggers has not arisen in our modelling of DEBSs.

Effect of Communication Differences In Kell calculi, the operational semantics need to take into consideration the restrictive channel communication, kell passivation rules and, in the case of Kell1 and Kell2, complex patterns in triggers, resulting in complicated reduction rules and behavioural equivalences. For example, while in kell-m there is only one kind of abstraction on channels ($a(c)$), in Kell1 and Kell2 there are three ($a^{\uparrow}(c)$, $a^{\downarrow}(c)$, $a(c)$), and in Kell3 there are five ($a^{\uparrow}(c)$, $a^{\downarrow}(c)$, $a^{\uparrow^K}(c)$, $a^{\downarrow^K}(c)$, $a(c)$).

Another problem in Kell calculi is the inconsistency between channel and kell abstractions. Specifically, arrow directives only apply to channel abstractions. As we show in Chapter 6, this is an issue when encoding the Kell calculi into a higher-order calculus with no concept of locality. The problem is that, because rules for kell and channel communication are different, kells cannot be easily modelled as regular processes that communicate via channels only.

In kell-m, on the other hand, the simplification and consistency of communication rules for channels and kells allow the representation of kells as processes evolving according to the following non-deterministic choice:

$$K[P] \equiv \overline{K}(P) \ + \ K[Advance(P)]$$

Where + is the π-calculus choice operator, and *Advance* advances the execution of P according to the semantics of kell-m (cf. Section 6.3.3 for a formal definition of *Advance*). $\overline{K}(P)$ is a higher-order output on channel K, a channel with the same name as the kell. P is the process within the kell. The result is simpler operational semantics and bisimulation theories for kell-m, when compared with Kell calculi. Moreover, because of its simpler semantics, an encoding of kell-m into higher-order π-calculus is possible as shown in Chapter 6, while a similar encoding is not yet known for any of the Kell calculi.

Typical control and modularization constructs can be modelled more succinctly in kell-m than in the Kell calculi. The communication rules in the Kell calculi have the undesirable effect of producing obfuscated models when using the Kell calculus to represent modularization concepts instead of code localization. The intuition behind the obfuscation is that, in the Kell calculi, processes encapsulating behaviour (e.g., abstract data types, classes, services), cannot be directly accessed by other client processes unless the client processes are at most one kell boundary away. Alternatively, requests from client processes need to be redirected via channels, one kell boundary at a time, until the request reaches the process encapsulating the behaviour. Similarly, responses from the encapsulating process need to be redirected all the way back to client processes. If no routing is done, the result is process code where the process encapsulating behaviour is replicated so that it is accessible to all clients, resulting in process duplication. If requests and responses are routed, the result is a specification where the routing processes and application specific processes are mixed together.

For example, recall the kell-m process used to stop the execution of a kell presented in Section 3.2:

$$\mathit{stop}(K) \diamond (K[x] \triangleright \mathbf{0})$$

Consider now the kell-m process, presented in Section 3.3.4, used to decide whether to execute a process P_t or a process P_f, based on which channel, t or f, is used:

$$\begin{array}{l} \mathit{casetf}(t, P_T, f, P_F) \diamond \mathbf{new}\ T,\ F\ (\\ \quad T[t() \triangleright P_T \mid \overline{\mathit{stop}}(F)] \mid \\ \quad F[f() \triangleright P_F \mid \overline{\mathit{stop}}(T)] \\ ) \end{array}$$

As previously mentioned, it is possible to model a conditional assuming another process that knows how to evaluate a particular condition and, given the two channels t and f, writes on t if the condition is *true*, or writes to f if the condition is *false*:

$$\mathbf{new}\ t,f\ (\overline{\mathit{cond}}(t,\ f) \mid \overline{\mathit{casetf}}(t, P_T, f, P_F))$$

For a particular condition *cond*, the above process models an **if else** statement:

$$\mathbf{if}\ \mathit{cond}\ \mathbf{then}\ P_T\ \mathbf{else}\ P_F\ \mathbf{fi}$$

In kell-m, such a construct only needs to be specified once and can be used in all kells as required:

$$K[\,T[\textbf{new}\ t,f\,(\overline{cond}(t,\,f)\mid \overline{casetf}(t,P_T,f,P_F))]]$$

In the Kell calculi, on the other hand, process code to route the requests to the process modelling the construct needs to be provided for every kell where the construct is used:

$$\begin{array}{l}
casetfFromK^{\downarrow}(t,P_T,f,P_F)\diamond\overline{casetf}(t,P_T,f,P_F)\mid\\
condFromK^{\downarrow}(t,f)\diamond\overline{cond}(t,\,f)\mid\\
K[\\
\quad casetfFromT^{\downarrow}(t,P_T,f,P_F)\diamond\overline{casetfFromK}(t,P_T,f,P_F)\mid\\
\quad condFromT^{\downarrow}(t,f)\diamond\overline{condFromK}(t,\,f)\mid\\
\quad T[\\
\qquad \textbf{new}\ t,f\,(\overline{condFromT}(t,\,f)\mid\overline{casetfFromT}(t,P_T,f,P_F))\\
\quad]\\
]
\end{array}$$

In some cases, as previously mentioned, the processes providing the services (accessible via *casetf* and *cond* in the example), may return a value that needs to be routed all the way down to the requester process, obfuscating even more the resulting Kell process.

The alternative to routing messages up and down kells is to replicate the process within all kells that need to access its services. Routing still may be required in some cases, for example when modelling variables (cf. Section 3.3.2), where a specific process holding the value of a variable needs to be accessed from several kells.

In summary, when compared with Kell calculi, kell-m communication rules facilitated the use of kells as a modularization construct. By not imposing restrictions on communications based on kell process location, it is possible to (when compared with Kell calculi) model succinctly the sharing of components between different kells. An alternate, more elaborated approach, based on an ownership sharing hierarchy is proposed by the creators of the Kell calculus in [86]. The main idea of the ownership sharing hierarchy is to introduce a new kind of name $*K$ representing a reference (i.e., pointer) to kell K. Hence, the following process:

$$K[P]\mid T[\,U[\,{*K}]]$$

is used to represent the equivalent process:

$$K[P]\mid T[\,U[\mathrm{K[P]}]]$$

The ownership sharing hierarchy complicates the operational semantics of the Kell calculi even further, and this line of work seems to have been abandoned by the creators of the calculus.

3.6.6 Comparison of Related Process Algebras

We conclude this section by comparing, in Table 3.1, the presented process algebras based on the following features:

- Communication: whether the communication is done via channels or other means, and whether the algebra supports synchronous or asynchronous communications.
- Higher-Order: whether processes can be transmitted as part of communications.
- Locality: what concept of location is supported by the algebra, if any, and whether structured locations are supported (i.e., whether a location can be within another location).
- Inter-Location Communication: what rules govern the communication between processes in different locations.
- Passivation: whether locations and their processes are concretions in the algebra that can be matched to process abstractions. Passivation allows the handling of sites and their processes as a unit with the purpose of suspending the execution of a site or adapting the site's running processes.
- Process Migration: whether a process can be moved from one site to another.
- Model Checker Support: whether tools exists for the model checking of systems specified in the given algebra.

3.7 Summary and Contributions

In this chapter we presented kell-m, a higher-order, asynchronous process algebra with hierarchical localities. Systems are represented in kell-m as processes executing in parallel. Processes communicate via channels and processes can be located within kells. Kells themselves can be located within other kells forming a kell containment hierarchy. Both channels and kells are identified by name. Names and processes can be transmitted as part of a channel communication.

Two kinds of process concretions are supported in the algebra: writes on channels ($\overline{a}(\widetilde{w})$), and executing kells ($K[P]$). The corresponding abstractions are channel patterns ($a(\widetilde{c})$) and kell patterns ($K[x]$) in triggers ($\xi \triangleright P$). When a channel concretion and channel abstraction match, a channel communication occurs, and the values written to the channel are received by the trigger where the abstraction is specified. When a kell concretion

	π-calculus	Dπ	Ambient Calculus	Join Calculus
Communication	sync. and async. channels	sync. and async. channels	sync. ambients; sync anonymous channels supported in boxed amb.	sync. and async. multi-channel join patterns;
Higher-Order	no	no	no	no in orig. join calculus; yes via OCaml functions in JoCaml
Locality	no	non-structured locations	structured ambients	structured sites in orig. join calc.; non-structured sited in JoCaml
Inter-Location Communication	n/a	no	yes, between neighbouring ambients only	yes, unrestricted
Passivation	n/a	no	no	no
Process Migration	n/a	yes, via **::** move operator	yes, to sibling ambient via **in** operator	yes in orig. join calc., via **go** operator; no in JoCaml
Automated Model Checker Support	yes	no	no	no

	M-Calculus	Kell Calculi	kell-m
Communication	sync. multi-channel join patterns	async. channels in Kell1 and Kell3; sync. channels in Kell2; multi-channel join patterns in Kell1 and Kell2	async. channels
Higher-Order	yes	yes	yes
Locality	structured cells	structured kells	structured kells
Inter-Location Communication	routing membrane processes	yes, within one kell boundary in Kell1 and Kell2; within kell containment branch in kell3	yes, unrestricted
Passivation	yes, of own cell by membrane process only	yes, within kell boundary	yes, unrestricted
Process Migration	no	yes, within kell boundary using passivation	yes, unrestricted using passivation
Automated Model Checker Support	no	no	yes

Table 3.1: Comparison of kell-m and Related Process Algebras

and kell abstraction are matched, the kell is passivated, and its process is available, via a variable, to the trigger where the kell abstraction is specified.

We introduced syntactic sugar to facilitate the use of kell-m and modelled basic control and modularization constructs. We call the resulting language hl-kell-m.

We presented the operational semantics for kell-m. These semantics specify how processes evolve as they communicate and are passivated. We defined two LTS and one reduction semantics, and used them to specify the behavioural equivalences for kell-m.

The main contributions of our work presented in this chapter are:

1. Simpler communication rules in kell-m when compared to previous kell-based algebras. Also in contrast to other kell-based algebras, in kell-m channel and kell abstractions and concretions are handled in a uniform way resulting in succinct operational semantics.

2. Extended LTS semantics that can be used by modal logics and automated tools, as shown in Chapters 4 and 6, to specify and verify kell containment conditions. The extended semantics subsume both, regular LTS and reduction semantics. We are not aware of any other kell algebras where such operational semantics have been developed.

3. Bisimilarity up to kell containment, a new behavioural equivalence defined based on the extended LTS semantics. This equivalence allows to check if two processes have the same kell containment structure without the need to compute bisimulation congruences.

4. Succinct representation of basic control constructs using kell-m when kells are used for modularization and process control instead of to represent physical locations.

We continue our presentation in the next chapter by introducing kμ, a logic for the specification of properties in systems represented using kell-m. Both kell-m and kμ are the basis of the system specification for DEBSs presented in Chapter 5 and the tools presented in Chapter 6.

Chapter 4

Formal Property Representation

Once a system has been modelled using kell-m, the operational semantics of kell-m can be used to study the evolution of the system. In particular, we are interested in being able to specify properties that must hold as a system evolves.

In this chapter we present kμ, a new formalism for the specification of properties of systems represented using kell-m. kμ is a modal temporal logic based on the extended operational semantics for kell-m presented in Section 3.5.3. For a given kell-m process, formulas specified in kμ impose conditions on the labelled transition system for the process. Conditions may specify kell containment requirements that must hold during transitions, and further kμ conditions that must hold after channel communications or kell passivations.

4.1 Background and Chapter Organization

The origins of kμ are shown in Figure 4.1. Arrows in the figure represent kind-of relationships, and dotted arrows represent extensions and modifications of a formalism.

Traditionally, modal and temporal logics have been used for the specification of properties of systems modelled using process algebras ([e.g., 81, 50, 171]). Modal logics deal with expressions qualified by operators that introduce, but are not limited to, concepts of necessity and possibility [75]. Examples of properties expressed in modal logics are *kell K may get passivated*, and *kell K must get passivated*. In these examples, the passivation of kell K is qualified by the operators *may* and *must*. These operators, called *modal operators* or *modalities*, are different from regular logic operators (e.g., *and*, *or*) because their truth value does not depend on the truth value of the expressions the modality is qualifying.

Temporal logics were introduced to computer science by Pnueli [135]. Temporal logics are modal logics where modalities express temporal requirements. Examples of temporal

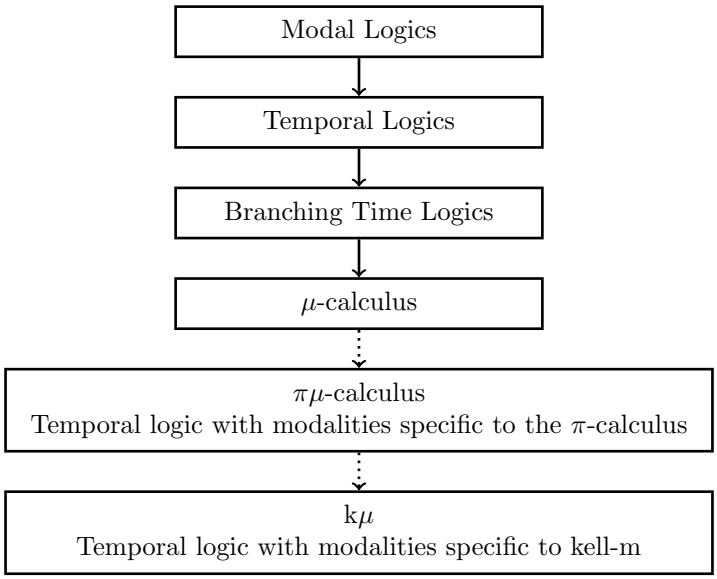


Figure 4.1: Lineage and categorization of kμ

modalities are **G**, **X**, and **U**. **G** specifies *it always be that*, and **X** specifies *next, it will be that*. **U** is a temporal operator over two expressions $\phi_1 \mathbf{U} \phi_2$, and it is used to specify that expression ϕ_1 *holds until a point in which* ϕ_2 holds. Other modalities can be derived from these modalities. For example, **F**, specifying *eventually it will be the case that*, is defined as $\mathbf{F}\phi \stackrel{def}{=} \neg\mathbf{G}\neg\phi$, where ϕ is the expression being affected by the modality.

When systems have different paths of evolution, some temporal logics support the quantification of these paths. Path quantification allows the expression of conditions such as *for all possible future computations* and *for some possible future computation* [57]. Temporal logics with quantification over paths are called *branching time logics*.

Branching time logics may impose conditions on the placement of path quantifiers in logic expressions. One of this logics is the Computation Tree Logic (CTL) [44]. In CTL, temporal modalities must be immediately governed by path quantifiers. For example, using $\forall$ and $\exists$ as universal and existential path quantifiers, $\forall(\phi_1 \mathbf{U}(\exists \mathbf{X} \phi_2))$ is a valid CTL expression, but $\forall(\phi_1 \mathbf{U}(\mathbf{X}\phi_2))$ is not. The reason is that $\mathbf{X}\phi_2$ is not immediately preceded by a path quantifier (example from [27]).

Most temporal logics can be defined in terms of the μ-calculus, another branching time logic [131, 58, 92, 57]. The novelty of the μ-calculus is in the use of least fixed-point (μ) and greatest fixed-point (ν) operators [158]. Along with μ and ν, the only other operators required to define other temporal modalities are path quantifiers ($\forall$, $\exists$), the next time operator **X**, and boolean operators ($\vee$, $\wedge$, $\neg$).

We now illustrate the use of fixed-point operators in μ-calculus. The explanation that follows is based on the presentation of fixed-points found in [57, 27, 29]. By definition a value x is a fixed-point of a function f if $f(x) = x$. Let us assume a non-empty set S with a relation $R : S \times S \rightarrow \{true, false\}$ that is reflexive ($aRa\ \forall a \in S$), antisymmetric ($\forall a, b \in S$, if aRb and bRa then $a = b$), and transitive ($\forall a, b, c \in S$, if aRb and bRc, then aRc). Such set S with relation R is called a *poset*. For a poset S, a least fixed-point is a fixed-point that is less or equal than all other fixed-points. Similarly, the greatest fixed-point is a fixed-point that is grater or equal than all other fixed-points.

For process algebras, with operational semantics based on labelled transition systems (LTSs) and a given process P, we define V as the set of process expressions (nodes) in the LTS. The set S can be specified as the set of all subsets (the power set) of V. The partial order relation R is the subset relation $\subseteq$. Expressions ϕ can be seen as the set of nodes in the LTS that satisfy ϕ. Abusing the notation, we can use ϕ to represent both a μ-calculus expression and the set of nodes that satisfy ϕ. Modalities such as $\exists\mathbf{X}$ can then be seen as functions $S \rightarrow S$. Therefore $\exists\mathbf{X}(\phi) = T$, where $T \in S$, and T is the set of nodes t such that there is a transition from t to t' in the LTS, and $t' \in \phi$. Within the context of LTSs for process algebras $\exists\mathbf{X}(\phi)$ specifies that, from the current node in the LTS, the process can evolve in one transition to a process expression on which ϕ holds.

The least fixed-point μ for a function f is then defined as:

$$\mu f = \bigcup_{\alpha<k} f^{\alpha}(\emptyset)$$

Where k is at worst $|V| + 1$ for finite V. A least fix point can then be computed as:

$$f(\emptyset) \cup f(f(\emptyset)) \cup f(f(f(\emptyset))) \cup \cdots$$

The greatest fixed-point is similarly defined:

$$\nu f = \bigcap_{\alpha<k} f^{\alpha}(V)$$

When using logic expressions instead of set expressions, for variable y and expression ϕ, we write $\mu y.\phi(y)$ and $\nu y.\phi(y)$, with:

$$\mu y.\phi(y) = \bigvee_{\alpha<k} \phi^{\alpha}(false)$$
$$\nu y.\phi(y) = \bigwedge_{\alpha<k} \phi^{\alpha}(true)$$

Greatest fixed-points are typically used to express *safety conditions*, while least fixed-points are used to express *liveness conditions*. Safety conditions allow the specification of *nothing bad happens* conditions, while liveness conditions allow the specification of *something good eventually happens* conditions.

Recall the only modalities needed in μ-calculus are $\forall\mathbf{X}$ and $\exists\mathbf{X}$. All the other temporal modalities can be derived. For example, $\forall\mathbf{G}\varphi$ is expressed as $\nu y.(\varphi \wedge \forall\mathbf{X}(y))$, which expands to:

$$\varphi \wedge \forall\mathbf{X}(\varphi \wedge \forall\mathbf{X}(\varphi \wedge \forall\mathbf{X}(\cdots\forall\mathbf{X}(\varphi \wedge \forall\mathbf{X}(\varphi \wedge \forall\mathbf{X}(\mathit{true})))\cdots)))$$

If using set expressions, $f(y) = \varphi \cap \forall\mathbf{X}(y)$, we have $\forall\mathbf{G}(\varphi) = \nu f$:

$$\varphi \cap \forall\mathbf{X}(\varphi \cap \forall\mathbf{X}(\varphi \cap \forall\mathbf{X}(\cdots\forall\mathbf{X}(\varphi \cap \forall\mathbf{X}(\varphi \cap \forall\mathbf{X}(V)))\cdots)))$$

Where $\forall\mathbf{X}(\phi) = \{t\}$, if t is the current node in the LTS, and for all transitions $t \rightarrow t'$, $t' \in \phi$. In particular $\forall\mathbf{X}(V) = \{t\}$ because, by definition of V, $\{t\} \in V$.

Using LTS semantics to represent the evolution of π-calculus processes, the $\pi\mu$-calculus extends the first-order μ-calculus with modalities for expressing conditions on channel communication actions [50].

Two kinds of modalities are introduced in the $\pi\mu$-calculus: *transition* modalities, and *io* modalities. The transition modalities are $\langle\alpha\rangle$ and $[\alpha]$, where α is a π-calculus communication action. Recall from Chapter 3, possible actions for the π-calculus are channel abstractions (e.g., a), concretions (e.g., $\overline{a}$), and channel communications (τ). A condition $\langle\alpha\rangle.\phi$ specifies that, from the current process expression, ϕ holds after at least one transition labelled with action α. A condition $[\alpha].\phi$ requires that, from the current process expression, ϕ holds after every transition labelled with action α.

The io modalities in the $\pi\mu$-calculus impose conditions on the names transmitted on channels. Having a represent a name, the io modalities are $a^{\rightarrow}$, $a^{\leftarrow}$, $a^{\nu\rightarrow}$, and $a^{\nu\leftarrow}$. Using l to represent a transition modality, $la^{\rightarrow}.\phi$ specifies that, besides condition $l.\phi$, the name a must have been received during l. Similarly, $la^{\leftarrow}.\phi$ specifies that, besides $l.\phi$, the name a must have been output during l. If a is restricted, then the modalities $a^{\nu\rightarrow}$ and $a^{\nu\leftarrow}$ are used instead.

kμ, the new formalism we present in this chapter, is a modal temporal logic for specifying properties on LTSs obtained with the extended operational semantics for kell-m defined in Section 3.5.3. As shown in Figure 4.1, kμ, is an extension of the $\pi\mu$-calculus. kμ extends the $\pi\mu$-calculus with kell-specific modalities and support for kell containment conditions.

We start the chapter with the specification of the syntax of kμ in Section 4.2. hl-kμ, a language based on kμ with syntactic constructs that improve the readability of kμ properties is introduced in Section 4.3. The semantics of kμ are then presented in Section 4.4, where we formalize what it means when we say a kμ property is satisfied by or holds for a kell-m specification. In Section 4.5 we discuss the kinds of properties expressible with kμ, and implications on model checking tools encoding kμ. Since kμ is derived from the μ-calculus, in Section 4.6 we use results on the complexity of model checking the

$$
\begin{array}{lll}
\mathcal{F} & ::= & \texttt{tt} \mid \texttt{ff} \mid \neg\mathcal{F} \mid \mathcal{C}.\mathcal{F} \mid \mathcal{F} \wedge \mathcal{F} \mid \mathcal{F} \vee \mathcal{F} \mid \langle\eta\rangle.\mathcal{F} \mid [\eta].\mathcal{F} \mid N(\widetilde{p}) \\
\mathcal{C} & ::= & a \ op \ b \mid a \in \widetilde{w} \mid |\mathcal{K}| = n \mid a \in \mathcal{K} \mid \neg\mathcal{C} \mid \mathcal{C} \wedge \mathcal{C} \mid \mathcal{C} \vee \mathcal{C} \\
\eta & ::= & \varphi \mid -\varphi \mid S \mid -S \\
\varphi & ::= & (\alpha, \gamma) \mid (\alpha_\tau, \gamma, \gamma) \\
\gamma & ::= & * \mid \mathcal{K} \mid \supseteq \mathcal{K} \mid \not\subseteq \mathcal{K} \mid I \\
N(\widetilde{p}) & ::= & \mathcal{F}\{\widetilde{p}/\widetilde{x}\} \text{ with } N(\widetilde{x}) \stackrel{def}{=} \mathcal{F} \\
op & ::= & = \mid \neq \mid > \mid < \mid \geq \mid \leq
\end{array}
$$

Figure 4.2: kμ Syntax

μ-calculus to argue about the complexity of model checking kell-m processes. We look at other formalisms used for the specification of properties of systems modelled using process algebras in Section 4.7, and conclude the chapter with a summary and a list of our contributions in Section 4.8.

4.2 Syntax

Properties in kμ are formulas $\mathcal{F}$ with the syntax specified in Figure 4.2. The syntax is inspired by the language implementation of the $\pi\mu$-calculus in the Mobility Model Checker [171]. Along with the term kμ *formula*, we also use the term kμ *condition* when referring to a kμ property.

`tt` represents true, `ff` represents false, $\neg\mathcal{F}$ is used to negate a condition specified by a formula. a and b represent names, n represents a number. $\mathcal{C}$ specifies comparison of values passed in communications, containment conditions on lists of names, containment conditions on kell sets, or checks on the size of a kell containment set.

Diamond $\langle\varphi\rangle$ and box expressions $[\varphi]$ impose conditions on transitions, where φ is an action condition. Diamond expressions are used to specify *in at least one transition where* φ conditions; while box expressions are used to specify *in every transition where* φ conditions. Hence, path quantification in kμ is implicit: existential for $\langle\ .\ \rangle$ modalities and universal for $[\ .\ ]$ modalities.

φ is a condition on a concretion or abstraction transition when φ is a pair (α, γ), and α represents an abstraction (i.e., $a(\widetilde{c})$, $K[x]$), or a concretion (i.e., $\overline{a}(\widetilde{w})$, $\overline{K}[P]$). γ represents a kell containment condition on the action α. Communication parameters in α (i.e., $\widetilde{c}$, $\widetilde{w}$, x, and P) can be variables and names. If they are names, they must match the names of the parameters in the transitions; if variables, they are instantiated with the corresponding parameter. For example, using uppercase for variables and lowercase for names, when

α is $\overline{a}(n,\ w)$, the only matched transitions in the LTS are those labelled with output on channel a of names n and w. When α is $\overline{a}(V_1, V_2)$, the matched LTS transitions are those which output two values on channel a, irrespectively of the actual values output. More examples will be provided once we introduce kell containment conditions and the other φ expressions.

φ is a condition on a τ transition when φ is a triple $(\alpha_\tau, \gamma, \gamma)$, and α_τ represents τ actions (i.e., $\overleftrightarrow{a}(\widetilde{w})$, $\overleftrightarrow{K}[P]$). Recall from Section 3.5.3, $\overleftrightarrow{a}(\widetilde{w})$ specifies the matching of channel abstraction $a(\widetilde{c})$ and concretion $\overline{a}(\widetilde{w})$, and $\overleftrightarrow{K}[P]$ specifies the matching of kell abstraction $K[X]$ and concretion $\overline{K}[P]$. The first γ in the triple is the kell containment condition for the abstraction; the second γ is the kell containment condition for the matching concretion.

We refer to kell containment conditions as *any* for $*$; *exactly* for $\mathcal{K}$; *at least* for $\supseteq \mathcal{K}$; *except* for $\nsubseteq \mathcal{K}$; and *instantiation* for I, where I is a variable.

For an action α, kell containment condition $*$ does not impose any requirements on the location of the action. Let us now assume α is located in a process P such that:

$$K_1[K_2[\cdots K_n[P]\cdots]]$$

And α is not inside a kell in P. With $\mathcal{K}$, a set of kell names, $\mathcal{K}$ holds if $\{K_1, K_2, ..., K_n\} = \mathcal{K}$. For example, $\{K_1, K_2\}$ holds if α occurs in R with $K_1[K_2[R]]$ or $K_2[K_1[R]]$. The same condition does not hold in $K_3[K_1[K_2[R]]]$. $\supseteq \mathcal{K}$ holds if $\mathcal{K} \setminus \{K_1, K_2, ..., K_n\} = \emptyset$, and $\nsubseteq \mathcal{K}$ holds if $\mathcal{K} \cap \{K_1, K_2, ..., K_n\} = \emptyset$. If a variable I is specified as a kell containment condition, the condition succeeds and I is instantiated to $\{K_1, K_2, ..., K_n\}$.

Consider the LTS shown in Figure 4.3 for a process P. Because we are using kell-m's extended LTS semantics, for every label δ_i in the LTS, δ_i may be: a pair α_i, κ_i, with α_i an action and κ_i its kell containment set; or $(\alpha_\tau, \kappa_i, \kappa_i')$, with α_τ specifying the channel or kell involved in the communication, κ_i the kell containment set for the abstraction, and κ_i' the kell containment set for the concretion.

Let us assume a boolean function *cs*, able to decide if a sequence of parameters in α or α_τ expressions is compatible with the actual parameters of an action specified in a LTS transition. *cs* will be formally defined in Section 4.4. Intuitively, a sequence of parameters $\widetilde{w}$ in α or α_τ is compatible with the actual parameters $\widetilde{d}$ of an action if both sequences have the same number of values and, positionally, every name in $\widetilde{w}$ is matched in $\widetilde{d}$.

Condition $\langle\overline{a}(\widetilde{w}), \{T\}\rangle.\mathcal{F}$ holds for P if there is at least one transition from P labelled with communication action $\overline{a}(\widetilde{d})$, the action is located only within kell T, names in $\widetilde{w}$ match names in $\widetilde{d}$, $cs(\widetilde{w}, \widetilde{d})$, and, after the transition, the formula $\mathcal{F}\{\widetilde{d}/\widetilde{w}\}$ holds for the resulting process expression. Formally, $\langle\overline{a}(\widetilde{w}), \{T\}\rangle.\mathcal{F}$ holds if $\exists P_i : P \xrightarrow{\delta_i} P_i$, such that $\delta_i = (\alpha_i, \kappa_i)$, with $\alpha_i = \overline{a}(\widetilde{d})$, $cs(\widetilde{w}, \widetilde{d})$, $\kappa_i = \{T\}$, and $\mathcal{F}\{\widetilde{d}/\widetilde{w}\}$ holds at P_i.

Conditions on transitions representing abstractions are similarly specified. For example $\langle a(\widetilde{c}), \{T\}\rangle.\mathcal{F}$ holds if $\exists P_i : P \xrightarrow{\delta_i} P_i$, such that $\delta_i = (\alpha_i, \kappa_i)$, with $\alpha_i = \overline{a}(\widetilde{e})$, $cs(\widetilde{c}, \widetilde{e})$, $\kappa_i = \{T\}$, and $\mathcal{F}\{\widetilde{e}/\widetilde{c}\}$ holds at P_i.

A condition on a τ transition, for example $\langle \overleftrightarrow{a}(\widetilde{w}), \{K\}, \{T\}\rangle.\mathcal{F}$, holds if $\exists P_i : P \xrightarrow{\delta_i} P_i$, such that $\delta_i = \overleftrightarrow{a}(\widetilde{d}), \kappa_a, \kappa_c$, with $\kappa_a = \{K\}$, $cs(\widetilde{d}, \widetilde{w})$, $\kappa_c = \{T\}$, and $\mathcal{F}\{\widetilde{d}/\widetilde{w}\}$ holds at P_i. A τ action on a kell holds when the specified kell is passivated, for example because the process in the kell is being adapted. In such case a property specifying that a kell K is being passivated while executing in L_c by a process executing in L_a is written as follows:

$$\langle \overleftrightarrow{K}(X), \{L_a\}, \{L_c\}\rangle$$

Such a property holds for the following process:

$$L_c[Q \mid K[P]] \mid L_a[R \mid K[X] \triangleright K[P']]$$

$[\overline{a}(\widetilde{w}), \emptyset].\mathcal{F}$ holds if, for every transition labelled with action $\overline{a}(\widetilde{d})$ not located within any kell, $\mathcal{F}\{\widetilde{d}/\widetilde{w}\}$ holds for the process expression after the transition. Formally, $[\overline{a}(\widetilde{w}), \emptyset].\mathcal{F}$ holds when $\forall P_i : P \xrightarrow{\alpha_i, \kappa_i} P_i$, if $\alpha_i = \overline{a}(\widetilde{w})$, $cs(\widetilde{d}, \widetilde{w})$, and $\kappa_i = \emptyset$, then $\mathcal{F}$ holds at P_i.

Similarly, a condition $[\overleftrightarrow{a}(\widetilde{w}), \{K\}, \emptyset].\mathcal{F}$, holds when $\forall P_i : P \xrightarrow{\delta_i} P_i$, if $\delta_i = [\overleftrightarrow{a}(\widetilde{d}), \kappa_a, \kappa_c]$, $cs(\widetilde{d}, \widetilde{w})$, and both, $\kappa_a = \{K\}$ and $\kappa_c = \{\}$, then $\mathcal{F}$ holds at P_i.

A property requiring adaptation of processes to occur only within certain localities is:

$$[\overleftrightarrow{K}(X), \{L_a\}, \{L_c\}]$$

If a kell K is passivated, the previous property requires the passivation to occur always while the passivated process executes within L_c by a process executing within L_a.

The use of variables as kell containment conditions is useful when requiring two different transitions to have the same kell containment conditions, without the need for knowing the actual kells. Hence $\langle \overline{a}(\widetilde{w}), I\rangle.\langle c(\widetilde{e}), I\rangle$ holds when there are consecutive transitions $P \xrightarrow{\overline{a}(\widetilde{d}), \kappa} Q \xrightarrow{c(\widetilde{g}), \kappa} R$. Notice the kell containment conditions κ are the same in both transitions.

The other possible $\langle\ .\ \rangle$ and $[\ .\ ]$ modalities deal with negation and sets of φ transition conditions. Having a process P and a set of communication condition pairs S, $\langle -\varphi\rangle.\mathcal{F}$ holds if there is at least one transition from P, for which φ does not stand, after which $\mathcal{F}$ holds. $\langle S\rangle.\mathcal{F}$ holds if there is at least one transition from P, for which a $\varphi \in S$ stands, after which $\mathcal{F}$ holds. $\langle -S\rangle.\mathcal{F}$ holds if there is at least one transition from P, for which a $\varphi \notin S$ stands, after which $\mathcal{F}$ holds. $[-\varphi].\mathcal{F}$ holds if $\mathcal{F}$ holds after every transition from P

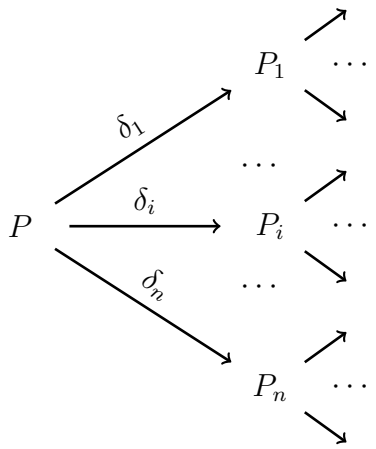


Figure 4.3: LTSs for $\langle\ .\ \rangle$ and $[\ .\ ]$ Examples

for which a φ' with $\varphi' \neq \varphi$ holds. $[S].\mathcal{F}$ holds if $\mathcal{F}$ holds after every transition from P for which $\varphi \in S$ holds. Similarly, $[-S].\mathcal{F}$ holds if $\mathcal{F}$ holds after every transition from P for which a condition $\varphi \notin S$ holds. We write $\langle - \rangle.\mathcal{F}$ for $\langle -\{\} \rangle.\mathcal{F}$, and $[-].\mathcal{F}$ for $[-\{\}].\mathcal{F}$.

Formulas can be named and parameterized using the syntax $N(\widetilde{x}) \stackrel{def}{=} \mathcal{F}$, where $\widetilde{x}$ are the formula parameters. $N(\widetilde{p})$ is an actual use of a named formula, where $\widetilde{p}$ are the values passed as formula parameters. Values can be channel and kell names, variables, and other named formulas. Naming of formulas allows recursive definitions. For example, the following formula has parameters C and K, and holds if, eventually, there is a read on given channel C within given kell K:

$$\mathit{ReadInKell}(C, K) \stackrel{def}{=} (\langle \overline{C}(\widetilde{c}), \supseteq \{K\} \rangle.\mathtt{tt}) \ \vee \ \langle - \rangle.\mathit{ReadInKell}(C, K)$$

Finally, notice there are no explicit quantifiers over variables. In the $\pi\mu$-calculus it is possible to express conditions $\forall c.\mathcal{F}$ requiring $\mathcal{F}$ to hold for every name c. In kμ variables are quantified implicitly depending on whether a diamond or box modality is used. Like paths, variables are existentially quantified in $\langle\ .\ \rangle$ and universally quantified in $[\ .\ ]$. For example, C is existentially quantified in $\langle a(C), * \rangle.\mathcal{F}$ and universally quantified in $[a(C), *].\mathcal{F}$. Since variable quantification is implicit in kμ, and only for values read or written, it is not possible to express conditions such as $\forall A : [A(C), *].\mathcal{F}$, meaning $\mathcal{F}$ *holds after every channel read*. However, variables can be used in place of channels and kells when the variables have been previously instantiated. For example, in the following property, variable *Rc* is instantiated with the third value passed in the communication on channel *subscribe*. The actual value corresponds to a channel on which a communication occurs right after the communication on channel *subscribe*:

$$\langle \overleftarrow{\mathit{subscribe}}(\mathit{Filter},\ \mathit{Callback},\ \mathit{Rc}) \rangle.\langle \overleftrightarrow{\mathit{Rc}}(S) \rangle$$

4.3 High-Level kμ

We introduce a few sugared constructs for kμ. The intention is to improve the readability of kμ properties. We name the resulting language hl-kμ and specify its grammar in Appendix A.

As usual, implication $\mathcal{F} \Rightarrow \mathcal{F}$ is defined from $\neg\mathcal{F}$ and $\mathcal{F} \vee \mathcal{F}$. We use *inert* to specify [-].`ff`. *inert* holds when, in the LTS, there are no more transitions from the current state.

We also introduce the following process definitions:

$$\begin{array}{rcl} \mathit{future}(N) & \stackrel{def}{=} & [-].(N \vee (\neg \mathit{inert} \wedge \mathit{future}(N))) \\ \mathit{future}_e(N) & \stackrel{def}{=} & \langle - \rangle.(N \vee \mathit{future}_e(N)) \\ \mathit{Eventually}(N) & \stackrel{def}{=} & N \vee \mathit{future}(N) \\ \mathit{Eventually}_e(N) & \stackrel{def}{=} & N \vee \mathit{future}_e(N) \end{array}$$

We use $\mathbf{F}(N)$, $\mathbf{F}_e(N)$, $\mathbf{E}(N)$, $\mathbf{E}_e(N)$ as shorthand notations for $\mathit{future}(N)$, $\mathit{future}_e(N)$, $\mathit{Eventually}(N)$, and $\mathit{Eventually}_e(N)$.

When no kell containment condition is specified, $*$ (*any*) is assumed. For example, we write $\langle a(c) \rangle.\mathcal{F}$ for $\langle a(c), * \rangle.\mathcal{F}$. Moreover, if no formula is specified after a $\langle\ .\ \rangle$ and $[\ .\ ]$ modalities, `tt` is assumed. Therefore condition $\langle a(c), * \rangle$.`tt` can be written as $\langle a(c) \rangle$.

Kell containment conditions for τ transitions are also optional, and $*$ is assumed for both abstraction and concretion. If only one kell containment condition is specified for a τ transition, it is assumed the condition applies to the abstraction, and $*$ any is assumed for the concretion. Therefore,

$$\begin{array}{rcl} \langle \overleftrightarrow{a}(c) \rangle & \equiv & \langle \overleftrightarrow{a}(c), *, * \rangle.\mathtt{tt} \\ \langle \overleftrightarrow{a}(c), \gamma \rangle & \equiv & \langle \overleftrightarrow{a}(c), \gamma, * \rangle.\mathtt{tt} \\ \langle \overleftrightarrow{a}(c), \gamma_a, \gamma_c \rangle & \equiv & \langle \overleftrightarrow{a}(c), \gamma_a, \gamma_c \rangle.\mathtt{tt} \end{array}$$

When specifying properties for methods in modules, communication channels can be specified as *module.method* and *inst::module.method*. Recall method modules are represented on channels named *module__method*, and receive as first parameter a method instance corresponding to a list of module variables (cf. Section 3.3.6). For example, we write $\langle \mathit{module.method}(\widetilde{ps}) \rangle$ for $\langle \mathit{module\_\_method}(\widetilde{ps}) \rangle$, and $\langle \mathit{inst::module.method}(\widetilde{ps}) \rangle$ for $\langle \mathit{module\_\_method}(\mathit{inst}, \widetilde{ps}) \rangle$.

Finally, property naming $\mathit{PropName}(\widetilde{ps}) \stackrel{def}{=} \mathcal{F}$, can be written as:

property $\mathit{PropName}(\widetilde{ps})$ { $\mathcal{F}$ }

4.4 Semantics

For a kμ formula $\mathcal{F}$, an *interpretation* $\mathcal{V}$ of formula parameters is a total function $\mathcal{V} : V \to N_K \cup N_F \cup N_V$. V are the parameter names in $\mathcal{F}$, N_k are kell-m names (of channels and kells), N_F are named formulas, and N_V are property variables, including variables in kell containment conditions. For simplicity we will assume no name clashes: $N_K \cap N_F \cap N_V = \emptyset$. Given a process P, a formula $\mathcal{F}$, and interpretation $\mathcal{V}$ of parameters in $\mathcal{F}$, we write $P \models_{\mathcal{V}} \mathcal{F}$ when $\mathcal{F}$ holds in P. Based on the LTS for P according to the extended semantics of kell-m (cf. Section 3.5.3), $\models_{\mathcal{V}}$ is defined inductively in Figure 4.4.

Because of property variables in the α and γ expressions, once a transition has occurred the resulting formula needs to be alpha-converted according to the instantiation of variables. Two kinds of instantiations can occur, depicted as $\mathcal{F}'$ and $\mathcal{F}''$, both specified in Figure 4.5. $\mathcal{F}'$ is used for τ transitions, while $\mathcal{F}''$ is used for abstractions and concretions.

Auxiliary functions used in the specification of the kμ semantics are defined in Figure 4.6. *kc* is a boolean function that, given a kell containment condition γ and kell containment set κ, decides if the kell containment condition holds. *cmp* is a boolean function able to decide if a condition action in a property formula applies to an action in a LTS transition. *cmp* uses boolean formula *cs* in its definition. *cs* determines if a sequence of parameter names and variables specified in a transition condition is compatible with a sequence of values specified in a LTS transition action. Finally, overloaded function *ps* extracts the list of parameters in an action condition or an action in a LTS transition.

Fixed point conditions are expressed using recursive formulas. For example, one may be interested in specifying condition *eventually, there is a write on channel a*:

$$F \stackrel{def}{=} (\langle \overline{a}(\widetilde{w}), \texttt{any} \rangle.\texttt{tt}) \vee \langle - \rangle.F$$

Which corresponds to the following least fixed-point definition:

$$\mu y.(\langle \overline{a}(\widetilde{w}), \texttt{any} \rangle.\texttt{tt} \ \vee \ \langle - \rangle.y)$$

Similarly, an example of a greatest fixed-point is condition *it is always possible to read on channel a*:

$$A \stackrel{def}{=} \langle a(\widetilde{c}), (\texttt{any} \rangle.\texttt{tt}) \ \wedge \ \langle - \rangle.A$$

Which corresponds to:

$$\nu y.(\langle a(\widetilde{c}), \texttt{any} \rangle.\texttt{tt} \ \wedge \ \langle - \rangle.y)$$

Another least fixed-point condition is *deadlock*, specified as:

$$deadlock \stackrel{def}{=} [-].\texttt{ff} \ \vee \ [-].deadlock$$

An example of a deadlocked process is **new** a $a(c) \triangleright P$. This process is deadlocked because it is waiting for communication on a private channel a no other process knows.

$P \models_\nu \mathtt{tt}$	always
$P \models_\nu \neg\mathcal{F}$	if $P \not\models_\nu \mathcal{F}$
$P \models_\nu \mathcal{C}.\mathcal{F}$	if $\mathcal{C} \ \wedge \ P \models_\nu \mathcal{F}$
$P \models_\nu \mathcal{F}_1 \wedge \mathcal{F}_2$	if $P \models_\nu \mathcal{F}_1 \ \wedge \ P \models_\nu \mathcal{F}_2$
$P \models_\nu \mathcal{F}_1 \vee \mathcal{F}_2$	if $P \models_\nu \mathcal{F}_1 \ \vee \ P \models_\nu \mathcal{F}_2$
$P \models_\nu \langle \alpha_\tau, \gamma_a, \gamma_c \rangle.\mathcal{F}$	if $\exists Q : P \xrightarrow{\alpha'_\tau, \kappa_a, \kappa_c} Q \ \wedge \ cmp(\alpha_\tau, \alpha'_\tau) \ \wedge$ $kc(\gamma_a, \kappa_a) \ \wedge \ kc(\gamma_c, \kappa_c) \ \wedge \ Q \models_\nu \mathcal{F}'$
$P \models_\nu \langle \alpha, \gamma \rangle.\mathcal{F}$	if $\exists Q : P \xrightarrow{\alpha', \kappa} Q \ \wedge \ cmp(\alpha, \alpha') \ \wedge$ $kc(\gamma, \kappa) \ \wedge \ Q \models_\nu \mathcal{F}''$
$P \models_\nu [\alpha_\tau, \gamma_a, \gamma_c].\mathcal{F}$	if $\forall Q : P \xrightarrow{\alpha'_\tau, \kappa_a, \kappa_c} Q$, $(cmp(\alpha_\tau, \alpha'_\tau),$ $(kc(\gamma_a, \kappa_a) \ \wedge \ kc(\gamma_c, \kappa_c))) \Rightarrow Q \models_\nu \mathcal{F}'$
$P \models_\nu [\alpha, \gamma].\mathcal{F}$	if $\forall Q : P \xrightarrow{\alpha', \kappa} Q,$ $(cmp(\alpha, \alpha') \wedge kc(\gamma, \kappa)) \Rightarrow Q \models_\nu \mathcal{F}''$
$P \models_\nu \langle -(\alpha, \gamma) \rangle.\mathcal{F}$	if $\exists Q : P \xrightarrow{\alpha', \kappa} Q \ \wedge$ $(\neg cmp(\alpha, \alpha') \vee \neg kc(\gamma, \kappa)) \ \wedge \ Q \models_\nu \mathcal{F}'$, or if $\exists Q : P \xrightarrow{\alpha_\tau, \kappa_a, \kappa_c} Q \ \wedge \ Q \models_\nu \mathcal{F}'$
$P \models_\nu \langle -(\alpha_\tau, \gamma_a, \gamma_c) \rangle.\mathcal{F}$	if $\exists Q : P \xrightarrow{\alpha'_\tau, \kappa_a, \kappa_c} Q \ \wedge \ (\neg cmp(\alpha_\tau, \alpha'_\tau) \vee$ $\neg kc(\gamma_a, \kappa_a) \ \vee \ \neg kc(\gamma_c, \kappa_c)) \ \wedge \ Q \models_\nu \mathcal{F}'$, or if $\exists Q : P \xrightarrow{\alpha, \kappa} Q \ \wedge \ Q \models_\nu \mathcal{F}'$
$P \models_\nu \langle \{\varphi_1, \varphi_2, ..., \varphi_n\} \rangle.\mathcal{F}$	if $P \models_\nu \langle \varphi_1 \rangle.\mathcal{F} \ \vee \ P \models_\nu \langle \varphi_2 \rangle.\mathcal{F} \ \vee \ \cdots \ \vee$ $P \models_\nu \langle \varphi_n \rangle.\mathcal{F}$
$P \models_\nu \langle -S \rangle.\mathcal{F}$	if $\exists \varphi : P \models_\nu \langle \varphi \rangle.\mathcal{F} \ \wedge \ \varphi \notin S$
$P \models_\nu [-\varphi].\mathcal{F}$	if $P \models_\nu [-\{\varphi\}].\mathcal{F}$
$P \models_\nu [\emptyset].\mathcal{F}$	always
$P \models_\nu [\{\varphi_1, \varphi_2, ..., \varphi_n\}].\mathcal{F}$	if $P \models_\nu [\varphi_1].\mathcal{F} \ \wedge \ P \models_\nu [\varphi_2].\mathcal{F} \ \wedge \ \cdots \ \wedge$ $P \models_\nu [\varphi_n].\mathcal{F}$
$P \models_\nu [-S].\mathcal{F}$	if $P \models_\nu [\varphi].\mathcal{F}$, with $\varphi \notin S$

Figure 4.4: kμ Semantics

$$\mathcal{F}' = \begin{cases} \mathcal{F}\{ps(\alpha'_\tau)/ps(\alpha_\tau)\} & \text{if } \gamma_a \notin N_v, \gamma_b \notin N_v \\ \mathcal{F}\{\kappa_a, ps(\alpha'_\tau)/\gamma_a, ps(\alpha_\tau)\} & \text{if } \gamma_a \in N_v, \gamma_b \notin N_v \\ \mathcal{F}\{\kappa_c, ps(\alpha'_\tau)/\gamma_c, ps(\alpha_\tau)\} & \text{if } \gamma_a \notin N_v, \gamma_b \in N_v \\ \mathcal{F}\{\kappa_a, \kappa_c, ps(\alpha'_\tau)/\gamma_a, \gamma_c, ps(\alpha_\tau)\} & \text{if } \gamma_a \in N_v, \gamma_b \in N_v \end{cases}$$

$$\mathcal{F}'' = \begin{cases} \mathcal{F}\{ps(\alpha')/ps(\alpha)\} & \text{if } \gamma \notin N_v \\ \mathcal{F}\{ps(\kappa, \alpha')/ps(\gamma, \alpha)\} & \text{if } \gamma \in N_v \end{cases}$$

Figure 4.5: Variable Instantiation in kμ Formulas

4.5 Potential versus Actual Communication

Given a LTS and a kμ formula, a kell-m checker verifies if the formula holds for the LTS by walking the LTS and evaluating conditions along the walk. These conditions are based on the labels in the LTS and impose requirements on the channels or kells involved in actions (abstraction, concretions, or τ), the actual values being passed in the actions (the $\widetilde{w}$ in $\overline{a}(\widetilde{w})$), and the kell location where the actions occur. The form the walk takes depends on the formula being verified. Recursive formulas have the potential of requiring a walk of the complete LTS, and nested recursive formulas may require several walks.

The LTS obtained with the extended semantics of Section 3.5.3, allows the specification of properties for both *potential* and *actual* communications. We say a process has the potential to communicate on channel or kell a, if there is an abstraction or concretion on a. For example, a process $a(c)$ has the potential to communicate on channel a. An actual communication is possible if both an abstraction and matching concretion occur in a process. Actual communications correspond to τ transitions in the extended LTS, while potential communications correspond to all other transitions.

When specifying properties using kμ, one has the choice of specifying conditions on transitions corresponding to potential communications, actual communications, or a combination. For example consider the following expression, previously introduced in Section 3.3.4, to represent conditionals:

$$\begin{array}{l} \mathit{casetf}(t, P_T, f, P_F) \diamond \textbf{new } T, F \, (\\ \quad T[t() \triangleright (P_T \mid \overline{\mathit{stop}}(F))] \mid \\ \quad F[f() \triangleright (P_F \mid \overline{\mathit{stop}}(T))] \\ ) \end{array}$$

When the previous process is composed with a process $\overline{\mathit{casetf}}(t, \overline{a}(r), f, \overline{b}(r))$, one may be interested in specifying that a write on channel a will happen after a read on channel t has occurred: $\mathbf{Ee}(\langle t() \rangle . \mathbf{Ee}(\langle \overline{a}(r) \rangle))$. Alternatively, one can further compose the process with

$$
\begin{array}{rl}
kc(*,\kappa) & \text{always} \\
kc(\mathcal{K},\kappa) & \text{if } (\kappa = \mathcal{K}) \\
kc(\supseteq \mathcal{K},\kappa) & \text{if } (\mathcal{K} \setminus \kappa = \emptyset) \\
kc(\not\supseteq \mathcal{K},\kappa) & \text{if } (\kappa \cap \mathcal{K} = \emptyset) \\
kc(V,\kappa) & \text{always}
\end{array}
$$

$$
\begin{array}{rl}
cmp(a(\widetilde{c}), g(\widetilde{e})) & \text{if } a = g \wedge cs(\widetilde{c},\widetilde{e}) \\
cmp(\overline{a}(\widetilde{w}), \overline{g}(\widetilde{d})) & \text{if } a = g \wedge cs(\widetilde{w},\widetilde{d}) \\
cmp(K[x], T[z]) & \text{if } K = T \wedge cs(x,z) \\
cmp(\overline{K}[x], \overline{T}[P]) & \text{if } K = T \wedge cs(x,P) \\
cmp(\overleftrightarrow{a}(\widetilde{w}), \overleftrightarrow{g}(\widetilde{d})) & \text{if } a = g \wedge cs(\widetilde{w},\widetilde{d}) \\
cmp(\overleftrightarrow{K}[x], \overleftrightarrow{T}[P]) & \text{if } K = T \wedge cs(x,P)
\end{array}
$$

$$
\begin{array}{rl}
cs(\widetilde{w},\widetilde{c}) & \text{if } |\widetilde{w}| = |\widetilde{d}| = 0 \\
cs(\widetilde{w},\widetilde{c}) & \text{if } \widetilde{w} = w,\widetilde{w}' \wedge \widetilde{c} = c,\widetilde{c}' \wedge \\
 & (w \in \mathcal{V} \vee (w \in N_k \wedge w = c)) \wedge cs(\widetilde{w}',\widetilde{c}')
\end{array}
$$

$$
\begin{array}{rcl}
ps(\overline{a}(\widetilde{w})) & \equiv & \widetilde{w} \\
ps(a(\widetilde{c})) & \equiv & \widetilde{c} \\
ps(K[x]) & \equiv & x \\
ps(\overline{K}[P]) & \equiv & P \\
ps(\overleftrightarrow{a}(\widetilde{w})) & \equiv & \widetilde{w} \\
ps(\overleftrightarrow{K}[P]) & \equiv & P
\end{array}
$$

Figure 4.6: Auxiliary Functions in kμ Semantics Definition

an expression writing on t and reading on a:

$$\overline{casetf}(t, \overline{a}(r), f, \overline{b}(r)) \mid \overline{t}() \mid a(d) \triangleright \mathbf{0}$$

Then, one can test for actual communications to occur in channel t and then on channel a:

$$\mathbf{Ee}(\langle \overleftrightarrow{t}() \rangle.\mathbf{Ee}(\langle \overleftrightarrow{a}(r) \rangle))$$

The first property requires potential for communication, the second property requires actual communication.

The type of conditions specified has implications on how model checking of kell-m processes is performed. If only potential communications are used, a formal model for a system can be typically model-checked by itself, without the need to compose the model with a

process expression that triggers the functionality represented in the model. When actual communications are used, it is frequently the case that such triggering functionality needs to be composed with the formal model being represented. For example, when modelling a server of some kind and specifying properties on actual communications, a client process requesting services may need to be provided. This client process would communicate with the server process to trigger the process code modelling service responses in the server process.

If properties are only expressed in terms of actual communications, the LTS for a given process expression can be reduced in size by only keeping τ transitions. The end result is a model checking process operating on a smaller LTS.

The size difference between a LTS including all transitions and a LTS including only τ transitions can be remarkable, even for simple processes. For example, consider the LTSs for processes P_1, P_2 and P_3 in Figure 4.7. For simplicity the kell containment sets are not shown in the transitions. Notice the higher order expression in P_3. Figure 4.8 shows the LTS for the composed process, $P_1|P_2|P_3$, when only actual communications are considered; Figure 4.9 shows the LTS for the same composed process considering all transitions. Because of space, only the names of the channels and kells involved in the actions are shown in the LTS in Figure 4.9.

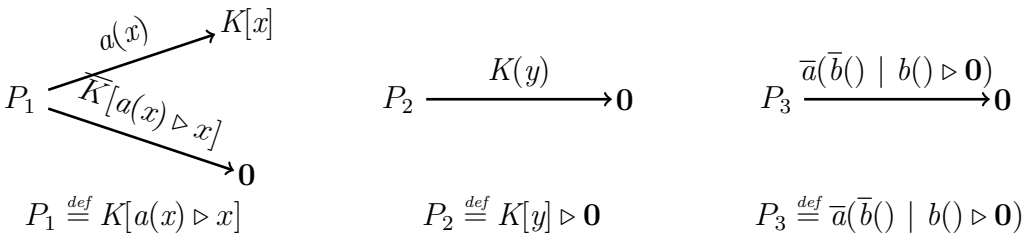


Figure 4.7: LTSs for Sample Processes

When actual communications are used the obtained LTS models the evolution of the process according to reduction semantics 3.4.2. When actual and potential for communication are considered, the LTS models the evolution according to LTS semantics 3.4.1.

4.6 Complexity

We provide an approximation of the complexity of verifying kμ formulas based on results on the complexity of verification for the μ-calculus. Given a LTS and a kμ formula, a kell-m checker verifies if the formula holds for the LTS by walking the LTS and evaluating

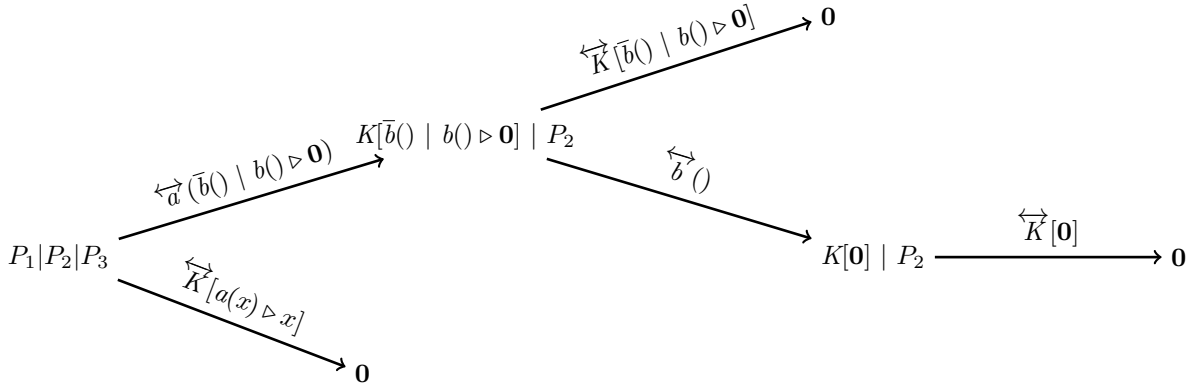


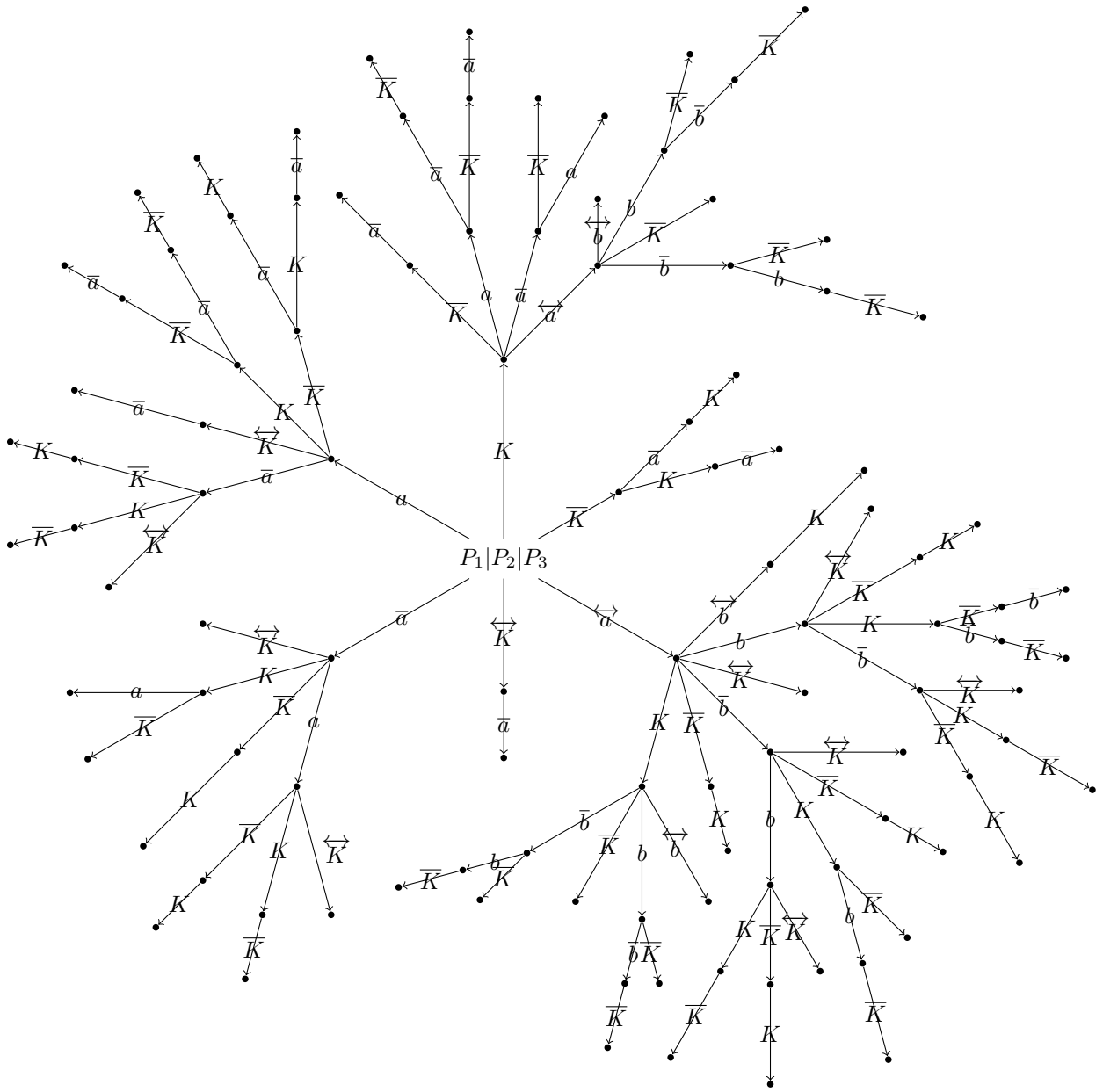
Figure 4.8: LTS with Actual Communications for Composed Process

conditions along the walk. Based on the labels in the LTS these conditions impose requirements on the channels or kells involved in actions (abstraction, concretions, or τ), the actual values being passed in the actions (the $\widetilde{w}$ in $\overline{a}(\widetilde{w})$), and the kell location where the actions occur. The form the walk takes depends on the formula being verified. Recursive formulas have the potential of requiring a walk of the complete LTS, and nested recursive formulas may require several walks.

We observe this is the same way verification of μ-calculus formulas is performed. μ-calculus formulas are verified by walking the LTS and depending on the kind of μ-calculus, by evaluating propositional (cf. [57]) or modal (cf. [27]) conditions expressed in terms of the transition labels. In the case of the μ-calculus, the form of the walk is determined by the fixed-points specified in the formula [57].

Since kμ is derived from the μ-calculus(cf. Section 4.1), the similarities between the two with regards to verification is not surprising. Recursive kμ formulas can be coded as fixed-point expressions in the μ-calculus. This follows from the fact recursive formulas in kμ are built the same way recursive formulas are built in MMC (cf. Sections 4.2 and 4.7.2), and it has been shown in [171] MMC recursive formulas correspond to fixed-point expressions.

As shown by Emerson in [57], the time complexity of verifying a formula $\mathcal{F}$ in a LTS *LTS* is $O((|\mathcal{F}| * (|S(LTS) + T(LTS)|)^{k+1})$; with $|\mathcal{F}|$ the size of $\mathcal{F}$, and $S(LTS)$ and $T(LTS)$ the number of states and transitions in *LTS*. k is a measure of how complex the fixed point expressions in the formula are. Specifically, k is the number of nested alternating least and greatest fixed-points when variables in the scope of the inner fixed-point are also in the scope of the outer fixed-point. In practice, formulas where $k \geq 2$ are not commonly



$P_1 \stackrel{def}{=} K[a(x) \triangleright x] \qquad P_2 \stackrel{def}{=} K[y] \triangleright \mathbf{0} \qquad P_3 \stackrel{def}{=} \overline{a}(\overline{b}() \mid b(\triangleright)\mathbf{0})$

Figure 4.9: LTS with Actual and Potential Communications for Composed Process

specified [26].

Using boolean equation systems (cf. [103]), time complexity of model checking for any formula of fixed-point nesting ≥ 2 can be reduced to $O(|\mathcal{F}|^2 * |S(LTS) + T(LTS)|)$, with space complexity of $O(|\mathcal{F}|^2 * |S(LTS)|)$ [107].

Assuming the complexity of evaluation of conditions along the LTS walk for both kμ and μ-calculus formulas is constant, or at least bound, one can code a kμ walk of the LTS as a μ-calculus walk on a similar LTS where, for each transition condition in the kμ formula, there is a similar transition condition in the μ-calculus formula. Based on this informal observation we argue the results on the complexity of verifying μ-calculus apply to the verification of kμ formulas.

The size of the formulas is typically very small compared with the size of the LTS. As illustrated by the example in the previous section (cf. Figures 4.8 and 4.9), the size of the LTS for kell-m processes grows combinatorially with the number of concretions, abstraction and τ actions. Also illustrated in the figures is the fact that higher-order expressions may also affect the size of the resulting LTS.

4.7 Related Work

We now present logics that have been proposed for the specification of properties in systems represented using process algebras. We start with the HML logic, one of the earliest logics proposed. We then look at the logic for property specification in the Mobility Model Checker. This is the logic from which the syntax of kμ is derived. We continue with the logic in the Mobility Workbench, and conclude with the logic proposed for the ambient calculus. With the exception of the logic for the ambient calculus, all other logics presented rely on the LTS operational semantics of the corresponding process algebra.

4.7.1 Hennessy-Milner Logic

Hennessy and Milner proposed HML, a logic for the verification of processes modelled using the Calculus for Communicating Systems CCS [81]. CCS is essentially the π-calculus with the restriction that only data can be passed in channel communications [114]. Recall, in the π-calculus, not only data but also channel names can be transmitted in communications. A formula in HML has the following syntax:

$$\mathcal{F} \ ::= \ \texttt{tt} \ | \ \texttt{ff} \ | \ \mathcal{F} \wedge \mathcal{F} \ | \ \mathcal{F} \vee \mathcal{F} \ | \ \langle \alpha \rangle . \mathcal{F} \ | \ [\alpha] . \mathcal{F}$$

Where α represent channel communications where only data is transmitted. Assuming a LTS for a CCS process P, $\langle\alpha\rangle.\mathcal{F}$ and $[\alpha].\mathcal{F}$ have the usual meanings:

$$\begin{array}{lcll} P & \models & \langle\alpha\rangle.\mathcal{F} & \text{if } \exists Q : P \xrightarrow{\alpha} Q \wedge Q \models \mathcal{F} \\ P & \models & [\alpha].\mathcal{F} & \text{if } \forall Q : P \xrightarrow{\alpha'} Q, (\alpha' = \alpha \Rightarrow Q \models \mathcal{F}) \end{array}$$

Since there is no recursion, it is not possible to express conditions such as *eventually there is a communication on channel a.*

4.7.2 Mobility Model Checker

As previously mentioned, the syntax of kμ is inspired by the implementation of the $\pi\mu$-calculus for the Mobility Model Checker (MMC) [171]. Systems are modelled in MMC using a variation of the first-order π-calculus. Since there is no locality in the π-calculus, the main difference between MMC and kμ, is the lack of kell and kell containment modalities in MMC. Also, because the first-order nature of its modelling formalism, properties in MMC can only be verified for first-order processes.

Formulas in MMC have the syntax specified in Figure 4.10. Recall possible actions α in the π-calculus have the form τ, $\overline{a}(\widetilde{c})$, or $a(\widetilde{c})$. S is a set of π-calculus actions, $\mathcal{V}$ is a set of names, $\mathcal{N}$ is a set of formula names, and $\mathcal{Z}$ is a set of formula variables. Formulas can be named using the syntax $\texttt{fDef}(\mathcal{N}(\widetilde{\mathcal{V}}), \mathcal{F})$. Recursion is allowed in the definition of named formulas.

Diamond and box modalities are specified with $\texttt{fDiam}(S, \mathcal{F})$ and $\texttt{fBox}(S, \mathcal{F})$. Set, set minus, and not-action box and diamond modalities have similar syntax. Only named formulas can be negated $\texttt{neg\_form}(\mathcal{N}(\widetilde{\mathcal{V}}))$. `pred` is used to compare names.

$$\begin{array}{rl} \mathcal{F} ::= & \texttt{tt} \mid \texttt{ff} \mid \texttt{neg\_form}(\mathcal{N}(\widetilde{\mathcal{V}})) \mid \texttt{pred(Cond}, \mathcal{F}) \mid \texttt{fAnd}(\mathcal{F}, \mathcal{F}) \mid \texttt{fOr}(\mathcal{F}, \mathcal{F}) \mid \\ & \texttt{fDiam}(\alpha, \mathcal{F}) \mid \texttt{fDiamMinus}(\alpha, \mathcal{F}) \mid \texttt{fDiamSet}(S, \mathcal{F}) \mid \\ & \texttt{fDiamSetMinus}(S, \mathcal{F}) \mid \texttt{fBox}(\alpha, \mathcal{F}) \mid \texttt{fBoxMinus}(\alpha, \mathcal{F}) \mid \texttt{fBoxSet}(S, \mathcal{F}) \mid \\ & \texttt{fBoxSetMinus}(S, \mathcal{F}) \mid \texttt{fDef}(\mathcal{N}(\widetilde{\mathcal{Z}}), \mathcal{F}) \mid \texttt{form}(\mathcal{N}(\widetilde{\mathcal{V}})) \end{array}$$

Figure 4.10: Syntax of Property Formulas in the Mobility Model Checker

As it is the case with kμ, in MMC process evolution paths are implicitly quantified. The quantification if *for all paths labelled with action* in the case of box modalities, and *in at least one path labelled with action* for diamond modalities.

$$
\begin{array}{rcl}
\mathcal{F} & ::= & \texttt{tt} \mid \texttt{ff} \mid a = b \mid a \# b \mid \mathcal{F}\&\mathcal{F} \mid \mathcal{F}|\mathcal{F} \mid \langle\alpha\rangle\mathcal{F} \mid [\alpha]\mathcal{F} \\
 & & \texttt{Sigma}\ a.\mathcal{F} \mid \texttt{Bsigma}\ a.\mathcal{F} \mid \texttt{Pi}\ a.\mathcal{F} \mid \texttt{exists}\ a.\mathcal{F} \\
 & & (\text{mu}\ \mathcal{N}(\widetilde{\mathcal{V}}).\mathcal{F})(\widetilde{\mathcal{V}}) \mid (\text{nu}\ \mathcal{N}(\widetilde{\mathcal{V}}).\mathcal{F})(\widetilde{\mathcal{V}}) \\
\alpha & ::= & t \mid a \mid {'a}
\end{array}
$$

Figure 4.11: Syntax of Property Formulas in the Mobility WorkBench

4.7.3 Mobility WorkBench

The Mobility Workbench (MWB) is a tool for analyzing mobile concurrent systems described using the π-calculus [160, 161]. MWB can be used to decide if two π-calculus expressions are bisimilar, or to verify if a condition holds for a given process. Processes being modelled must be closed: all names must be bound in the process expression. An implementation of the μ-calculus with π-calculus modalities allows the specification of μ-calculus formulas in MWB [19].

The syntax of μ-calculus formulas in MWB is based on a predecessor of the $\pi\mu$-calculus [49], and is specified in Figure 4.11. `tt` stands for *true*, and `ff` for *false*. $a = b$ is name equality, and $a \# b$ is name inequality. & is used for *and*, and | for *false*. Actions α can be t for τ, the name of a channel a for abstractions and, for concretions, the name of the channel is prefixed with a quote character $'a$. Operators mu and nu are used to specify least and greatest fixed-point conditions.

Name quantifiers are supported. `Pi` a represents *for all names* a; `exists` a represents *for some name* a. `Sigma` and `Bsigma`, are used to specify conditions on the names being transmitted. The MWB formula $\langle a\rangle$`Sigma` $c.\mathcal{F}$, is equivalent to the μ-calculus expression $\langle a\rangle c^{\rightarrow}.\mathcal{F}$ (cf. Section 4.1 for a brief description of the μ-calculus and its syntax). Similarly, the MWB formula $'\langle a\rangle$`Sigma` $c.\mathcal{F}$ is equivalent to the μ-calculus expression $\langle\overline{a}\rangle c^{\leftarrow}.\mathcal{F}$. `Bsigma` is used instead of `Sigma` to specify that a name read or written is restricted. The corresponding μ-calculus expressions when `Bsigma` is used are $\langle a\rangle c^{\nu\rightarrow}.\mathcal{F}$ and $\langle\overline{a}\rangle c^{\nu\leftarrow}.\mathcal{F}$.

In contrast to MWB, in kμ there are no distinct modalities for the specification of channels and data being transmitted. For example, $\langle a(c), \texttt{any}\rangle.\mathcal{F}$ in kμ specifies a read on channel a, of one name c. The equivalent MWB expression requires the use of two modalities: $\langle a\rangle$ and `Sigma` c. Other differences between MWB and kμ include the explicit syntax for fixed-point expressions in MWB, the use of explicit name quantifiers in MWB, and the lack of support in MWB for locality modalities.

$$\mathcal{F} \quad ::= \quad T \mid \mathbf{0} \mid \neg\mathcal{F} \mid \mathcal{F} \vee \mathcal{F} \mid \mathcal{F}|\mathcal{F} \mid \mathcal{F} \triangleright \mathcal{F} \mid n[\mathcal{F}] \mid \mathcal{F}@n \mid n\textcircled{R}\mathcal{F} \mid \mathcal{F} \oslash n \mid \diamond\mathcal{F} \mid \star\mathcal{F} \mid \forall x\mathcal{F}$$

Figure 4.12: Syntax of Property Formulas in the Ambient Calculus

4.7.4 Logic for Mobile Ambients

Recall the ambient calculus is a process algebra where the basic communication abstraction is not channel communication but the movement of ambients from one ambient to another [34] (also, cf. Section 3.6.2 for a description of the ambient calculus). Cardelli and Gordon propose a logic for the specification of properties of processes represented using the ambient calculus [36]. An encoding of the logic and model checker tool is described in [147]. In contrast to the other logics reviewed in this section, this logic is not derived from the μ-calculus, and recursion and fixed-points are not supported. The syntax of the logic is specified in Figure 4.12.

T is used to represent *true*. n represents ambient names in the formulas. $\mathbf{0}$ holds if the process is the null process. $\mathcal{F}_1|\mathcal{F}_2$ holds for a process $P|Q$ if $\mathcal{F}_1$ holds for P and $\mathcal{F}_2$ holds for Q. $\mathcal{F}_1 \triangleright \mathcal{F}_2$ holds for a process P when for all processes P', if $\mathcal{F}_1$ holds for P', then $\mathcal{F}_2$ holds for $P|P'$. $n[\mathcal{F}]$ holds for a process $n[P]$ if $\mathcal{F}$ holds for P. $\mathcal{F}@n$ holds for a process P, if $\mathcal{F}$ holds for $n[P]$. $n\textcircled{R}\mathcal{F}$, called *revelation*, holds for a process P if there is a process P' such that $P = \mathbf{new}\ n\ P'$, and $\mathcal{F}$ holds for P'. $\mathcal{F} \oslash n$, called *revelation adjunct* or *hiding*, holds for a process P, if $\mathcal{F}$ holds for $\mathbf{new}\ n\ P$. $\diamond\mathcal{F}$, named *sometime*, is a modality that holds for process P if P can be weakly reduced (i.e., in any number of reductions) to a process P', and $\mathcal{F}$ holds for P'. $\star\mathcal{F}$, called the *somewhere* modality, holds if there is an ambient P', at any depth inside P, and $\mathcal{F}$ holds for P'. $\forall x\mathcal{F}$ holds for a process P if, for every name m, when x is replaced by m in P, $\mathcal{F}$ holds for the resulting process $P\{m/x\}$.

The logic allows the specification of properties such as *there is an ambient a_p with a child ambient a_c*, and *process $P|Q$ executes within ambient a*. The first property imposes an ambient containment requirement, equivalent to the kell containment conditions that can be expressed in kμ. The second property imposes a requirement on how processes are structured by parallel composition and cannot be expressed in kμ. This is because kμ is based on the kell-m assumption that only communication actions are observable, as proposed by Sangiorgi and Milner [117]. In particular, parallel composition is not observable. Also, since the null process $\mathbf{0}$ is not observable, in kμ it is not possible to specify a condition *process* $\mathbf{0}$ *is in kell* K, while in the logic for the ambient calculus the equivalent condition $\mathbf{0}$ *is in ambient* K can be specified.

With the exception of kμ, the logic for mobile ambients and derived logics are the only logics, we are aware of, that allow the specification of locality (process containment)

modalities for systems represented using a process algebra. In contrast to kμ, the mobile ambient logic does not rely on the LTS of the algebra for its semantics. Instead, as illustrated in [33] for a logic derived from the logic for mobile ambients, sets of processes closed under structural congruence can be used. The process sets are called *property sets*. Given a formula, a process that meets the formula is constructed. Any process for which the formula holds must be in the property set for the constructed process.

4.8 Summary and Contributions

In this chapter we presented kμ, a modal temporal logic for the representation of properties for systems specified using kell-m. Specifically, kμ is a formalism to specify conditions on the LTSs obtained using the extended kell-m semantics presented in Section 3.5.3. kμ is an extension of the $\pi\mu$-calculus, a logic with π-calculus modalities.

Formulas in kμ have the form $\langle\varphi\rangle.\mathcal{F}$, $[\varphi].\mathcal{F}$, $\langle-\varphi\rangle.\mathcal{F}$, $[-\varphi].\mathcal{F}$, $\langle-\{\varphi_1,\varphi_2,...\}\rangle.\mathcal{F}$, and $[-\{\varphi_1,\varphi_2,...\}].\mathcal{F}$, with $\mathcal{F}$ a formula, and φ a condition on a transition. Conditions on transitions specify the action being executed by the process and, optionally, a kell containment condition restricting the location of the process executing the action. When φ is negated, $-\varphi$, the kμ formula applies to transitions where the transition condition φ does not hold.

When the condition φ on a transition is within a diamond modality, $\langle\varphi\rangle.\mathcal{F}$, the formula holds if there is at least one transition from the current process such that the transition meets the conditions imposed by φ, and formula $\mathcal{F}$ holds for the resulting process. When within a box modality, $[\varphi].\mathcal{F}$, $\mathcal{F}$ must hold for every process for which a transition from the current process meets the conditions imposed by φ.

For sets of transition conditions, $\langle\{\varphi_1,\varphi_2,...\}\rangle.\mathcal{F}$ is equivalent to $\langle\varphi_1\rangle.\mathcal{F}\vee\langle\varphi_2\rangle.\mathcal{F}\vee...$, and $[\{\varphi_1,\varphi_2,...\}].\mathcal{F}$ is equivalent to $[\varphi_1].\mathcal{F}\wedge[\varphi_2].\mathcal{F}\wedge....$

Properties in kμ can specify conditions on the potential for communication, actual communications via channel and kell passivation, or the combination of potential and actual communications. When adaptive behaviour is modelled using kell passivation, kμ can be used to specify conditions on the adaptation of processes executing within kells.

Properties can be recursive, allowing the specification of fixed-point conditions. We introduced syntactic constructs to facilitate the specification of kμ formulas. To the best of our knowledge, kμ is the only formalism that has been proposed for the specification of properties in a kell-based process algebra.

Because kμ is defined based on the LTS obtained with kell-m's extended semantics, it is possible to develop tools for the automated verification of kμ formulas. Receiving as input a kell-m process and a kμ formula such tools must be able to represent the evolution

of the process as a LTS using kell-m extended semantics and, based on the semantics of kμ presented in this chapter, verify if the LTS conforms to the given kμ formula.

Finally, formalisms for property specification that have been proposed for π-calculus and π-calculus derived algebras only allow the specification of properties based on the actions taken as the process evolves. With kμ it is also possible to impose conditions on the location at which the actions take place. Although locations in kell-m may correspond to physical locations in the case of formal models for DEBSs, kell-m localities can be used to represent other types of locations. For example, when modelling the architecture of a program, one can use kells to identify actions that occur within a relevant architectural location (e.g., module, class, or aspect in the case of Aspect Oriented Programming [90]).

Chapter 5

Case Studies

In this chapter we use hl-kell-m to develop formal models representing DEBSs. These models support the specification of DEBS properties previously proposed in the area using other formalisms. Based on the extended LTS semantics of kell-m, it is now possible to verify these properties on the models developed. This is contrast to work based on trace semantics where verification is only possible for given executions of a system and not for the system itself.

We also show how new properties, based on the locality features provided by kell-m and the ability to passivate kells, can now be specified and verified.

The models presented in this chapter are based on standards proposed for DEBSs and documentation for specific systems.

5.1 Background and Chapter Organization

Liveness and safety properties for DEBSs were identified by Fiege and Mühl [119, 69]. The specification of these properties was based on trace semantics without developing a formal model for the behaviour in DEBSs. With trace semantics, execution traces record the operations performed by components in a system. The identified properties specify the types of traces well-behaved DEBSs must produce.

With trace semantics the study of the behaviour exhibited by DEBSs is limited to validating that a given trace satisfies a given property. Even when the trace satisfies a property there is no guarantee the system which generated the trace always satisfies the property. This is because in the work of Fiege and Mühl there is no model or mechanism that can be used to generate all possible traces of a system.

In this chapter we present models for DEBSs. We show how properties previously identified for DEBSs can still be specified, and how new properties constraining the location of actions, passivation of kells and, in general, application-level abstractions not expressible with trace semantics can now be specified in these models. In contrast with previous work, tools can be developed to verify if a property holds on a given model. One can think of kell-m and its extended LTS semantics as the mechanism that allows the generation of all possible traces for a model. Furthermore, while in trace semantics properties can only be specified for the operations for which traces are produced, with kell-m all actions specified in the models are exposed to property specifications.

Each DEBS provides a set of calls that publisher and subscriber components invoke to interact via events (cf. Chapter 1). This set of calls is known as the *DEBS API*. At the very least a DEBS API must allow subscribers to specify the events of interest and publishers to announce events to the system. Our approach to specifying DEBS behaviour is to model DEBSs as systems that support a DEBS API and behave according to the DEBS event model presented in Chapter 2.

The formal models in this chapter specify features commonly found in DEBSs. Features that typically vary among different systems are parameterized in the models. Because of our process-based approach to the specification of DEBSs, the parameterization of features is done by assuming processes at predetermined channels model specific details for a system of interest. For example, delivery of events is parameterized via a *delivery* channel. When subscribers to be notified of an event in the system are identified, a *callback* channel registered by the subscriber and the event of interest are given to the process at the *delivery* channel. A model for a specific system can then be produced by composing the specification of a DEBS API compatible with the event model of the DEBS of interest, and processes specifying the functionality parameterized in the DEBS API model. These processes model functionality specific to the DEBS of interest:

$$\mathit{debs\_model} \mid \mathit{SystemSpecificFeature}_1 \mid \mathit{SystemSpecificFeature}_2 \mid \cdots$$

We start in Section 5.2 by modelling systems that follow the Common API. The Common API is a DEBS API that has been proposed as the standard API for DEBSs. Most DEBSs readily support the Common API, or can be easily modified to support it [133]. In Section 5.3 we model Rebeca, a DEBS with support for scopes. A scope is a mechanism for structuring DEBSs and applications. The modelling of scopes showcases the use of kell-m to represent advanced features proposed for DEBSs. In Section 5.4 we show how kells can be used to model the hierarchical structure of NaradaBrokering. This model showcases the use of kell-m to model the architectural aspects of DEBSs. Related work is presented in Section 5.6. We summarize the chapter and list our contributions in Section 5.7.

5.2 Common API

Although there is no standard DEBS API supported by each DEBS, a *Common API* has been proposed by Pietzuch et al. [133]. This Common API consists of two other APIs: a *Core API*, and an *Optional API*. DEBSs following the Core API are referred to as *simple DEBSs* [68, 69, 133]. The Core API contains the basic calls required in the subscription and announcement of events. The Optional API extends the Core API by providing calls for DEBSs requiring the advertisement of the events to be published (cf. Section 2.2).

The APIs only list the calls, parameters, and returned values. No data types or implementation details are specified. How the calls are implemented in a specific DEBS determines the event model provided by the system (cf. Chapter 2).

As shown by Pietzuch et al., the APIs can be supported by well-known DEBSs with little effort. This is the main reason for our use of the APIs: in this chapter we model DEBSs as systems providing a DEBS API and behaving according to a DEBS event model.

The calls in the Core API are:

$$\texttt{subscribe}(\texttt{filter}, \texttt{callback}, \texttt{ttl}) \rightarrow \texttt{subscription}$$
$$\texttt{unsubscribe}(\texttt{subscription})$$
$$\texttt{publish}(\texttt{event})$$

`filter` is an expression determining the events of interest to a component. When an event of interest has been published, it is communicated to the interested component via a `callback` routine. `ttl` is a time-to-live or expiration date determining for how long a subscription should be kept in the system. A `subscribe` call returns a `subscription` which, depending on the system, could be an object or handle.

The process *coreapi_debs* in Figure 5.1 represents generic functionality for DEBSs supporting the Core API. The process takes as arguments channels *subscribe*, *unsubscribe*, *publish*, and *deliver*. With the exception of the *deliver* channel, the other channels correspond to the calls in the API.

The list *subsc* in *coreapi_debs* stores the active subscriptions managed by the DEBS. *sem* is a binary semaphore (cf. Section 3.3.3), used to guarantee exclusive access when the list of subscribers is being modified.

Assuming a *subscription* process, specified in Figure 5.2, when a subscription request is received a subscription s is created, returned, and added to *subsc*. The synchronous variable assignment ($:=_s$, cf. Section 3.3.2) is used to be able to release the semaphore after the list has been updated. An unsubscribe request removes the given subscription from *subsc*.

When an event is published, each subscription must decide if the event is to be delivered to the subscriber. Given our process-based approach, *filter* is a channel where a process,

```
process coreapi_debs(subscribe, unsubscribe, publish, deliver) {
  fresh sem (
    sem() |
    var subsc := [ ] in (
      subscribe(filter, callback, ttl, rc) ◇ (
        fresh s (
          subscription(s, filter, callback, ttl)
          |
          sem() ▷ (subsc :=_s @cons(s, *subsc)) ▷ (rc(s) | sem())
        )
      )
      |
      unsubscribe(s) ◇ (
        sem() ▷ (subsc :=_s @del(*subsc, s)) ▷ sem()
      )
      |
      publish(e) ◇ (
        foreach s in *subsc do @s(deliver, e) done
      )
    )
  )
}
```

Figure 5.1: Core API Specification

```
process subscription(notify, filter, callback, ttl) {
  notify(deliver, event) ◇ (
    if (@filter(event) and @ttl()) then
      @deliver(callback, event)
    fi
  )
}
```

Figure 5.2: Subscription Process for Core API

capable of identifying if a published event should be delivered to a component, is waiting for filtering requests. Similarly, *callback* is a channel where a subscriber process is waiting for event notifications. *ttl* is a channel representing a time-to-live or expiry time for the subscription. *event* is a channel where a process representing an event can be accessed.

Delivery of an event is performed by a process at channel *delivery*. The specification of such a process depends on the actual system being modelled. A trivial delivery process is:

$$deliver(callback,\ event) \diamond @callback(event)$$

Other possible delivery process specifications are discussed in Section 5.2.1 below.

5.2.1 Specification of Event Model Variations

The event model of a system determines how events are defined, how they are announced to other components, how components manifest their interest in events, and how events are delivered to interested components. In Chapter 2 we characterized typical event model variations within DEBSs. In this section we show how these diverse event models are supported by the *coreapi_debs*.

As required by the DEBS event model, in the *coreapi_debs* process the kinds of events are not predetermined, events are explicitly announced, interested components must register their interest in events, dynamic event binding is modelled using the *subsc* list in each DEBSs, and events are delivered to all interested components. These features are supported by *coreapi_debs* as follows.

Variable Event Vocabulary. The kinds of events that components announce is not predetermined by the system (cf. Section 2.2). In *coreapi_debs* events are represented as channels. Processes at the channels model the event implementation details. Whether events must be declared or not before they can be published and subscribed to is not explicitly specified in *coreapi_debs*. If event declaration is required, this can be specified as a separate process. For example a process *event_support* may provide a means for creating and registering new kinds of events. Using kμ one can verify all events published have been previously registered. An specification for a DEBS is then obtained by composing *coreapi_debs* and the *event_support* model.

Event advertisement is not supported by the Core API and, therefore, it is not supported by its kell-m specification *coreapi_debs*. *optapi_debs* (cf. Section 5.2.6), the kell-m specification of the Optional API, does support event advertisement.

Event Attributes. Data attributes can be associated to events in most DEBSs (cf. Section 2.3). Because events are specified as processes in the formal models, no conditions are imposed on the structure of the events.

Recall the *temperature* module presented in Section 3.3.6 for storing a temperature at a given latitude and longitude. Assuming the *coreapi_debs* specification, a temperature event can be published by a process such as:

$$t{:}temperature(22, 43, 80) \triangleright @publish(t)$$

When modelling a DEBS where events are represented as list of strings, for example JEDI [47, 48] (cf. Section 2.2), a temperature event can be published as:

$$@publish([\text{“}temperature\text{”}, \text{“22”}, \text{“43”}, \text{“80”}])$$

Notice there is no need to change the *coreapi_debs* specification irrespectively of whether or not events are structured.

Dynamic Event Binding. An event binding determines which components are to be announced when an event is published (cf. Section 2.4). In *coreapi_debs* a list is used to store subscriptions. When an event is announced, the event bindings are dynamically computed by iterating through the list of subscriptions looking for subscriptions with filters matching the event published. New subscriptions can be added to the list and subscriptions can be removed as well. In *coreapi_debs* publishers cannot specify the intended recipients of the events and subscribers cannot specify the publishers of the events.

Attribute Binding. When events are published, only a subset of the attributes in the event may be delivered to a subscriber (cf. Section 2.5). We are aware of only one system, Padres [88], supporting this feature. It is necessary to alter *coreapi_debs* to support attribute binding. Before delivering an event, @*deliver*(*callback, event*) in the model, the event can be passed to a channel *abinding* previously registered by the subscription. The process at *abinding* can alter the event by keeping the attributes of interest only. The altered event can be returned and then delivered in-lieu of the original event. For *coreapi_debs*, the specification of this feature is depicted in Figure 5.3.

Event Announcement. Publisher components must explicitly announce the events by invoking a `publish` operation (cf. Section 2.6). This is specified in *coreapi_debs* via the *publish* channel and the process waiting for communications at that channel. Safety properties for *coreapi_debs* later specified in Section 5.2.2 require events to be published before they are delivered. As previously mentioned, no addressing information (i.e., which subscribers should be notified) is specified by the components announcing the events.

```
process subscription(notify, filter, callback, ttl, abinding) {
   notify(deliver, event) ◇ (
      if (@filter(event) and @ttl()) then
         @abinding(event)(event_ab) ▷ @deliver(callback, event_ab)
      fi
   )
}
```

Figure 5.3: Subscription Process with Attribute Binding for Core API

Event Subscription. Subscriber components are required to inform the system of the kinds of events they want to be notified about (cf. Section 2.7). In the *coreapi_debs* this feature is specified using a channel *subscribe*. Safety properties for *coreapi_debs* require that no component is notified of events before the component has subscribed to them (cf. Section 5.2.2).

As part of the subscription components specify a filtering channel. A process at a filtering channel must be able to decide if an event is of interest. This decision may be based on the type of the event, event attribute conditions, and contextual information for the event. No conditions are imposed by *coreapi_debs* with regards to the specification of these filters.

When temperature events are modelled using the *temperature* module presented in Section 3.3.6, a filter for temperatures greater than a value *tmin* can be specified as:

```
process tempfilter {
   filter(e, rc) ◇ fresh t, f (
      rc̄(t, f) | if (*e::temperature.temp > tmin) then t̄() else f̄() fi
   )
}
```

Interest on such events is modelled by the following process:

$$tempfilter() \mid callback(e) \diamond P \mid @subscribe(tempfilter,\ callback,\ ttl)(s) \triangleright \cdots$$

Event Delivery. Events are delivered to all interested components. There are variations with regards to the delivery semantics and ordering guarantees among different DEBSs (cf. Section 2.8). In *coreapi_debs* the delivery semantics are parameterized using a channel *delivery*: when an event is published the interested subscribers are identified and, for each identified subscriber, the event and callback channels are passed to the process at channel *delivery*. The process at channel *delivery* receiving the event and callback determines the specific delivery semantics of the system of interest.

A safety property for *coreapi_debs* requires the process waiting for requests at channel *delivery* to be invoked for all interested subscribers of a published event (cf. Section 5.2.2).

A DEBS with best-effort delivery semantics can be specified as:

$$delivery(callback,\ event)\ \diamond\ @callback(event)$$

At least once semantics can be specified using acknowledgments:

$$\begin{array}{l} delivery(callback,\ event)\ \diamond\ (\\ \quad \textbf{fresh}\ t\ (\\ \quad\quad @settimer(t,\ n)\ \triangleright\ (\\ \quad\quad\quad @callback(event)()\ \triangleright\ @disable(t)\ \mid \\ \quad\quad\quad t()\ \triangleright\ @delivery(callback,\ event) \\ \quad\quad) \\ \quad) \\ ) \end{array}$$

When an event is received for delivery, a new timer t is set to n and the *callback* is invoked. If there is no acknowledgment from the callback, the timer goes off and delivery of the event is attempted again. Because the timer may go off even if an acknowledgment is received, an event may be notified more than once. Delivery of the event may be attempted a determined number of times.

DEBSs guaranteeing ordered delivery of events typically implement an specific protocol (e.g., Gryphon [20, 173], also cf. Section 2.8). Such a protocol needs to be specified using kell-m to model these systems. By assuming events are timestamped when they are published, one can use queues in kell-m to model DEBSs with varying delivery guarantees. Even quality aspects such as network reliability can be modelled. We sketch how such specifications can be done in kell-m as follows. For simplicity, we will assume exclusive access to the lists in the processes. In a complete specification semaphores should be used to guarantee exclusive access.

Event queues in a system can be represented as two ordered lists of events. A head list contains the oldest events, based on their publication time. The amount of time between the first and last elements in the list does not exceed a configurable threshold. A tail list contains the events, ordered by publication date (we will generalize the ordering criteria later), outside the threshold. Hence, the time difference between the first element of the head list and any element in the tail list exceeds the threshold.

If we call the threshold Δt, when Δt is close to 0, the threshold models a reliable network with low latency where an event e_1, published at time t_1 (publisher's time), arrives to a queue of events before an event e_2 with publication time t_2. If $t_2 - t_1 > \Delta t$, we can say that e_1 occurred prior to e_2. If $\Delta t = 0$, the network is reliable with no latency (or the same

latency for all components in the DEBS). In this case the head list will contain a very small number of events when compared to the tail list. This is because it is possible to determine with high certainty if one event happened before other. Hence, the head list contains the next events to process for which we cannot determine with certainty which one occurred before the other. As Δt increases, the network is assumed to be less reliable and latency fluctuations greater. A high value of Δt models a DEBS with no order guarantees.

To generalize this idea of two ordered lists, with list membership determined by a threshold, we assume a process that knows the value of Δt, and given two events e_1 and e_2, tells us whether it is able to determine if e_1 occurred before e_2, or if e_2 occurred before e_1, or if it is not possible to determine which one occurred first. This process can consider other factors besides the publication time of the events, for example event priorities. We assume the existence of such a process, which we call cmp_e. We also assume this cmp_e process returns a value < 0 if it is able to determine that the first event passed occurred prior to the second; > 0 if the second occurred prior to the first; and 0 if it is not able to make a decision.

To simplify the specification, we write:

$$\textbf{if } (e_1 \ cmp_e \ e_2 > 0) \textbf{ then } P \textbf{ elsif } (e_1 \ cmp_e \ e_2 < 0) \textbf{ then } Q \textbf{ else } R \textbf{ fi}$$

instead of,

$$@cmp_e(e_1, e_2)(order) \ \diamond \ \textbf{if } (order > 0) \textbf{ then } P \textbf{ elsif } (order < 0) \textbf{ then } Q \textbf{ else } R \textbf{ fi}$$

We also assume the existence of a process cmp_t, similar to cmp_e, but that decides if an event was published before another one, *assuming perfect conditions*. cmp_t will be used to order events within a list, and cmp_e will be used to decide to which list an event should be added.

A process *equeue* specifying queuing of events in DEBSs as previously described is shown in Figure 5.4. The process at channel *push* adds a new event to the head list *hl* or the tail list *tl* depending on the result of comparing, using cmp_e, the new event against the events in the head list *hl*. The addition of the new event may cause events to be moved from *hl* to *tl*. This happens when the new event is added to the head list and the threshold between the new event and the events to move to *tl* is exceeded. Auxiliary process at channel *addtohl* is in charge of moving events from the head list to the tail list when the new event is added to the head list. The process at channel *addtohl* returns the new head list. The process at channel $cons_s$, presented in Section 3.3.5, is used to construct a sorted list.

The process at channel *pop* randomly takes one of the events in the head list *hl* and returns it. The event to pop is selected randomly because there is no way to determine with certainty which event in the head list occurred first. The event that is returned is

```
process equeue(empty, push, pop) {
  fresh addtohl var hl := [ ], tl := [ ] in (
    empty(rc) ◇ isempty(*hl, rc)
    |
    push(e) ◇ (
      if @isempty(*hl) then
        hl := @cons(e, *hl)
      elsif (e cmp_e @pos(*hl, 1) < 0) then
        var rhl := @reverse(*hl) in (hl :=_s [ ] ▷ @addtohl(e, rhl))
      elsif (e cmp_e @pos(*hl, 1) = 0) then
        hl := @cons_s(e, *hl, cmp_t)
      else
        tl := @cons_s(e, *tl, cmp_t)
      fi
    )
    |
    pop(rc) ◇ (
      @random(*hl)(l, e) ▷ (
        @rearrange(l, *tl)(nhl, ntl) ▷ ( hl :=_s nhl ▷ tl :=_s ntl ▷ rc(e) )
      )
    )
    |
    addtohl(e, rhl) ◇ (
      match rhl with
        [ ] ▷ 0
        or e_h :: es ▷
          if (e cmp_e e_h = 0) then
            hl :=_s @cons_s(e_h, *hl, cmp_t) ▷ addtohl(e, es)
          else
            tl :=_s @cons_s(e_h, *tl, cmp_t) ▷ addtohl(e, es)
          fi
    )
}
```

Figure 5.4: Event Queue for Ordered Delivery

also removed from the list. After the removal of the event, events in the tail list that fall within the ordering criteria for events in the head list (as determined by cmp_e) are moved from the tail list to the head list by the process at channel *rearrange*, shown in Figure 5.5.

```
rearrange(hs, ts, rc) ◇ (
    match hs with
        [ ] ▷ (
            match ts with
                [ ] ▷ rc̄([ ], [ ])
                or t :: tr ▷ rearrange̅(@cons(t, [ ]), tr, rc)
        )
        or h :: hr ▷ (
            match ts with
                [ ] ▷ rc̄(hs, tr)
                or t :: tr ▷ (
                    if (h cmp_e t = 0) then
                        rearrange̅(@append(hs, [t]), tr, rc)
                    else
                        rc̄(hs, ts)
                    fi
                )
        )
)
```

Figure 5.5: Rearrangement of Events After Event Removal

The following process randomly selects and removes an element from a list:

```
random(ss, rc) ▷ new R, rc′ (
        R[foreach s in ss do rc′̄(s)] |
        rc′(s) ▷ (stop̄(R) | rc̄(@del(ss, s), s))
)
```

A process *equeue* can be associated with each subscription when specifying DEBS with ordered delivery.

Event Persistence. When an event is published and a subscribed component is not available, some DEBSs store the event until the subscriber is available or until a time-to-live associated with the event (cf. Section 2.9). This can be specified in kell-m by queuing events and by allowing subscribers to poll for missed events when they reconnect

to the system. Alternatively, when a subscriber reconnects the events can be pushed to the subscriber. Such functionality requires acknowledgment of events received upon notifications, similarly to how ordering guarantees were previously specified in this section.

In some systems such as REBECA, events are stored by specialized *history components* and can be replayed on demand [119]. This can be specified using kell-m by representing history components as subscribers of the events they store. These history subscribers are also registered to be notified of *replay event requests* events. When such a request event is received by a history subscriber, the component re-publishes the events it has stored. Care needs to be taken so that only the component requesting the replay is notified of the replayed events. This can be accomplished by creating a subscription on the fly, just for the replaying of historical events, and having the requesting component as the only subscriber.

Event Notification. The provision method for event notifications can be push, pull, and push-pull (cf. Section 2.10). The notification method modelled by *coreapi_debs* is push. For pull and push-pull it is necessary to queue events. In general a queue is required per subscription. When pull and push-pull are supported, callbacks invoked when event notifications need to remove the notified events from the subscription event queue. Therefore a channel must be provided as part of the subscription to allow subscriber to pop events from the subscription queue.

5.2.2 Safety and Liveness Properties

We show how basic safety and liveness properties, previously proposed for DEBSs [119, 69], can be specified when the *coreapi_debs* process is used as the model of a DEBS. We specify two safety properties: events are not delivered to uninterested subscribers, and delivery of an event to a subscriber never occurs prior to the subscription and publication. We also specify a liveness property requiring published events to be notified of all subscribed components.

We start by specifying a property holding when a time-to-live for a subscription is still active:

$$\textbf{property}\ c\_active(Ttl)\ \{\ \mathbf{E}_e(\langle \overleftrightarrow{Ttl}(Rc)\rangle.returns\_true(Rc))\ \}$$

We use uppercase for variables. $\mathbf{E}_e$, existential eventually, was defined in Section 4.3, and holds if the formula received as its only parameter holds in at least one transition from the current process, or in the future as the process evolves.

For a subscription, *c_active* holds if the process at *Ttl* returns *true*. Recall from Section 3.3.4, booleans in kell-m are represented by two channels. If communication occurs on the

first channel, *true* is assumed, and if communication occurs on the second channel, *false* is assumed instead. In *returns_true*, the channel corresponding to *true* is represented with variable T:

$$\textbf{property}\ returns\_true(Rc)\ \{\ \mathbf{E}_e(\langle \overleftrightarrow{Rc}(T,F)\rangle.\mathbf{E}_e(\langle \overleftrightarrow{T}()\rangle))\ \}$$

Properties *c_active* and *returns_true* require actual communications to occur in the process. Actual communications correspond to τ transitions in the LTS representing the evolution of the process. Potential for communication corresponds to LTS transitions for abstractions and concretions. The following property uses potential for communication to specify the condition under which a time-to-live is active:

$$\textbf{property}\ l\_active(Ttl)\ \{\ \mathbf{E}_e(\langle \overline{Ttl}(Rc)\rangle.l\_returns\_true(Rc))\ \}$$

where *l_returns_true* is defined as:

$$\textbf{property}\ l\_returns\_true(Rc)\ \{\ \mathbf{E}_e(\langle \overline{Rc}(T,\ F)\rangle.\mathbf{E}_e(\langle \overline{T}()\rangle))\ \}$$

In this chapter we use actual communication conditions when specifying properties. As discussed in Section 4.5, the LTS is smaller when using actual communications allowing for faster automated verification of properties.

We now specify a property *c_interested*, holding when a given event *Event* is of interest to a subscription filter *Filter*:

$$\textbf{property}\ c\_interested(Filter,\ Event)\ \{\ \mathbf{E}_e(\langle \overleftrightarrow{Filter}(Event,\ Rc)\rangle.returns\_true(Rc))\ \}$$

The following safety property holds if the system cannot evolve such that an event is delivered to an active subscription without interest in the event:

$$\begin{array}{l}\textbf{property}\ c\_of\_interest\_only()\ \{\\ \quad \neg\mathbf{E}_e(\langle \overleftrightarrow{subscribe}(Filter,\ Callback,\ Ttl,\ Rc)\rangle.\\ \qquad (\mathbf{E}_e(\langle \overleftrightarrow{deliver}(Callback,\ Event)\rangle) \wedge \neg c\_interested(Filter,\ Event))\\ \quad)\\ \}\end{array}$$

We assume the callback and filter channels are unique to each subscription.

The following property should not hold on processes representing DEBSs. It specifies an event delivery occurring before a subscription to the event:

$$\begin{array}{l}\textbf{property}\ c\_delivery\_before\_subsc()\ \{\\ \quad \langle \overleftrightarrow{deliver}(Callback,\ Event)\rangle\ \vee\\ \quad \langle -\overleftrightarrow{subscribe}(Filter,\ Callback,\ Ttl,\ Rc)\rangle.c\_delivery\_before\_subsc()\\ \}\end{array}$$

Recall from Section 4.4, a modality $\langle -\overleftrightarrow{subscribe}(Filter,\ Callback,\ Ttl,\ Rc)\rangle.\mathcal{F}$ holds for a process P if $\exists Q : P \xrightarrow{\alpha,\kappa} Q$, $\alpha \neq \overleftrightarrow{subscribe}(Filter,\ Callback,\ Ttl,\ Rc)$, and $\mathcal{F}$ holds for Q.

Similarly, a delivery of an event cannot happen before the publication of the event:

$$
\begin{array}{l}
\textbf{property}\ c\_delivery\_before\_pub()\ \{ \\
\quad \langle \overleftrightarrow{deliver}(Callback,\ Event)\rangle\ \vee \\
\quad \langle -\overleftrightarrow{publish}(Event)\rangle.c\_delivery\_before\_pub() \\
\}
\end{array}
$$

When an event is published, DEBSs attempt to notify all subscribed components. The following safety property holds if the publication of an event and the interest of a subscription on the event imply the event is delivered to the subscriber, unless the subscriber unsubscribes or the time-to-live for the subscriber is exceeded:

$$
\begin{array}{l}
\textbf{property}\ c\_all\_notified()\ \{ \\
\quad (\mathbf{E}_e(\langle \overleftrightarrow{publish}(Event)\rangle)\ \wedge\ \mathbf{E}_e(\langle \overleftrightarrow{subscribe}(Filter,\ Callback,\ Ttl,\ SubRc)\rangle. \\
\qquad \mathbf{E}_e(\langle \overleftrightarrow{SubRc}(S)\rangle.c\_interested(Filter,\ Event)))\ \Rightarrow \\
\quad (\mathbf{E}_e(\langle \overleftrightarrow{deliver}(Callback,\ Event)\rangle)\ \vee\ \mathbf{E}_e(\langle \overleftrightarrow{unsubscribe}(S)\rangle)\ \vee \\
\qquad \mathbf{E}_e(\langle \overleftrightarrow{Ttl}(TtlRc)\rangle.\neg returns\_true(TtlRc))) \\
\}
\end{array}
$$

The following safety property for DEBS with exactly-once delivery (cf. Section 2.8), requires delivery of events to occur no more than once for each event:

$$
\begin{array}{l}
\textbf{property}\ c\_once\_notification()\ \{ \\
\quad \neg\mathbf{E}_e(\langle \overleftrightarrow{deliver}(Callback,\ Event)\rangle.\mathbf{E}_e(\langle \overleftrightarrow{deliver}(Callback,\ Event)\rangle)) \\
\}
\end{array}
$$

In the previous property we assume the callback specified in each subscription is unique. We also assume each event is uniquely identified and published once.

Given the semantics of kell-m and kμ, a major contribution of our work is the ability to verify these safety and liveness properties without the need to execute actual implementations of the DEBSs and applications being modelled. This is in contrast to other models proposed for DEBSs ([e.g., 119, 69]), where one must run the system of interest and collect and analyze execution traces to confirm if the particular system execution met the properties being verified. Moreover, by relying on execution traces, there is no way to guarantee a future execution will meet a given property. We return to the shortcomings of trace semantics in Section 5.6.

5.2.3 Unordered Delivery

Few DEBSs guarantee events are notified in the same order they are delivered (cf. Section 2.8). The following property specifies events may not be delivered in the order they are published:

$$
\begin{array}{l}
\textbf{property}\ \textit{c\_unordered\_notification}() \ \{ \\
\quad \mathbf{E}_e(\langle \overleftrightarrow{\textit{publish}}(E_1) \rangle . \\
\qquad \mathbf{E}_e(\langle \overleftrightarrow{\textit{publish}}(E_2) \rangle . \\
\qquad\quad \mathbf{E}_e(\langle \overleftrightarrow{\textit{deliver}}(C_2, E_2) \rangle . \mathbf{E}_e(\langle \overleftrightarrow{\textit{deliver}}(C_1, E_1) \rangle)) \\
\qquad) \\
\quad) \\
\}
\end{array}
$$

A DEBS does not provide ordering guarantees if the publication of E_1 followed by the publication of E_2 may be followed by the delivery of E_2 prior to the delivery of E_1.

5.2.4 Kell Containment Properties

We now look at properties beyond the traditional liveness and safety properties specified for DEBSs. It is possible to use kells and kell containment conditions when formalizing features of interest in a system. For example, one may be interested in guaranteeing events of a certain kind are only published by components executing on a certain location. Using kells to model locations, such a property can be specified as:

$$
\begin{array}{l}
\textbf{property}\ \textit{c\_from\_site\_only}(\textit{Site, Event}) \ \{ \\
\quad [\overleftrightarrow{\textit{publish}}(\textit{Event}), K_r, K_w].(\textit{Site} \in K_w) \ \wedge \ [-].\textit{c\_from\_site\_only}(\textit{Site, Event}) \\
\}
\end{array}
$$

Or,

$$
\textbf{property}\ \textit{c\_from\_site\_only}(\textit{Site, Event}) \ \{ \ \neg \mathbf{E}_e(\langle \overleftrightarrow{\textit{publish}}(\textit{Event}), K_r, \not\subseteq \{\textit{Site}\} \rangle) \ \}
$$

For example, *c_from_site_only*(*waterloo, event*) holds if all events *event* are only published by components within kell *waterloo*. In *c_from_site_only*, instead of specifying:

$$
[\overleftrightarrow{\textit{publish}}(\textit{Event}), K_r, K_w].(\textit{Site} \in K_w)
$$

one may be tempted to write:

$$
[\overleftrightarrow{\textit{publish}}(\textit{Event}), *, \supseteq \{\textit{Site}\}]
$$

which is equivalent to:

$$[\overleftarrow{\overrightarrow{publish}}(Event), *, \supseteq \{Site\}].\texttt{tt}$$

But such a condition just establishes that `tt` must hold for every event published at *Site*, not that every event *Event* should be published at *Site*.

In general, when it is required that every process performing an action α within a kell K must meet a condition $\mathcal{F}$ the formula to specify is:

$$[\alpha, *, \supseteq \{K\}].\mathcal{F}$$

When every action α must occur within a kell K, the condition to specify is:

$$[\alpha, *, k_w].(K \in K_w).$$

A site may be forbidden from publishing events. Such a property would be specified as:

property *c_not_at_site*(*Site*) {
$[\overleftarrow{\overrightarrow{publish}}(Event), K_r, K_w].(Site \notin K_w) \ \wedge \ [-].c\_not\_at\_site(Site)$
}

Publishing components not at *Site* may need to meet extra condition $\mathcal{F}$:

property *c_extra_cond_if_not_site*(*Site*) {
$[\overleftarrow{\overrightarrow{publish}}(Event), K_r, \not\subseteq \{Site\}].\mathcal{F} \ \wedge \ [-].c\_extra\_cond\_if\_not\_site(Site)$
}

Event dependency conditions can also be specified based on location. For example, consider the case when publications of E_1 at *Site* are always followed by a publication of event E_2:

property *c_chained_events*(E_1, E_2, *Site*) {
$\mathbf{E}_e(\langle\overleftarrow{\overrightarrow{publish}}(E_1), K_r, \supseteq \{Site\}\rangle \Rightarrow \mathbf{E}(\langle\overleftarrow{\overrightarrow{publish}}(E_2)\rangle))$
}

When events are structured, sometimes it is necessary to include kell-m expressions exposing the data in the events. This allows properties such as *c_from_site_only* above to identify events of interest. For example, a kell-m process exposing the data for temperature events represented as instances of a *temperature* module (cf. Section 3.3.6) is:

process *expose_temp*(*T, Rc*) {
$\overline{Rc}$(**T::temperature.temp*, **T::temperature.lat*, **T::temperature.lon*)
}

Such a process needs to be invoked for every temperature event published. In general, one can modify the *publish* call to receive the event and a channel able to expose the event's data. Alternatively, a unique process able to expose the data for all events in the system can be assumed. This generic process could be invoked by the process at channel *publish*. The actual approach will depend on the system being specified and the properties to verify.

Once the model has been modified to expose event information, a property *temp_at_location* can identify, given an event, locality requirements for the publishing process:

$$\begin{array}{l}\textbf{property}\ \mathit{temp\_at\_location}(T,\ Lt,\ Ln)\ \{\\ \quad \mathbf{E}_e(\langle \overleftarrow{\mathit{expose\_temp}}(T,\ Rc)\rangle.\mathbf{E}_e(\langle \overleftrightarrow{Rc}(\mathit{Temp},\ \mathit{Lat},\ \mathit{Lon})\rangle.(\mathit{Lat} = Lt \wedge \mathit{Lon} = Ln)))\\ \}\end{array}$$

A more accurate check for location can be done calculating distances based on the given latitude and longitude coordinates. Such a computation is possible with the tools described in Chapter 6, where predicates can be specified as conditions on values passed in communications. For simplicity, in this example we check for exact latitude and longitude coordinates.

We then modify property *c_from_site_only* to receive as parameter the coordinates of interest:

$$\begin{array}{l}\textbf{property}\ \mathit{c\_from\_site\_only}(\mathit{Kell},\ \mathit{Event},\ \mathit{Lat},\ \mathit{Lon})\ \{\\ \quad [\overleftarrow{\mathit{publish}}(\mathit{Event}), K_r, K_w].(\mathit{temp\_at\_location}(\mathit{Event}, \mathit{Lat}, \mathit{Lon}) \Rightarrow \mathit{Kell} \in K_w)\ \wedge\\ \quad [-].\mathit{c\_from\_site\_only}(\mathit{Kell},\ \mathit{Event},\ \mathit{Lat},\ \mathit{Lon})\\ \}\end{array}$$

Similar properties can be specified to restrict the location of subscribed components.

Although the ability to localize processes exists in all kell-based formalisms (cf. Section 3.6.5), we are not aware of other works where it is possible to specify and verify locality conditions for kell processes.

5.2.5 Kell Passivation Properties

Besides the ability to specify properties imposing conditions on the locations of the actions, another novel aspect of our work is the ability to specify properties on the passivation of kells. For example, one may be interested in specifying that after a given event is received by a subscribed component, the subscribed component is passivated and moved to another location. Such a property is useful when specifying services that migrate based on the availability of resources. A service may receive an event indicating that the resources at its current location (for example a computer) are running low. The reaction of the subscribed event is then to migrate to another computer where resources are available.

Consider the following property:

$$
\begin{array}{l}
\textbf{property}\ \mathit{service\_migration}(\mathit{Callback},\ \mathit{Service},\ \mathit{Event})\ \{ \\
\quad [\overleftrightarrow{\mathit{Callback}}(\mathit{Event}), \supseteq \{\mathit{Service}, \mathit{Computer}_1\}, *].(\\
\qquad \mathbf{E}(\langle\overleftrightarrow{\mathit{Service}}[X], \supseteq \{\mathit{Computer}_2\}, *\rangle.(\mathit{Computer}_1 \neq \mathit{Computer}_2)) \\
\quad) \ \wedge\ [-].\mathit{service\_migration}(\mathit{Callback},\ \mathit{Service},\ \mathit{Event}) \\
\}
\end{array}
$$

where *Service* is the kell for the migrating service and *Callback* is the callback registered by the service for the notification of events representing low resources. For simplicity we assume the value provided in the property parameter *Event* matches such events.

The property *service_migration* holds for systems where the service *Service* is notified of a low resource event *Event* while executing at computer $\mathit{Computer}_1$ and its reaction is to move to a computer $\mathit{Computer}_2$. Notice the action $\overleftrightarrow{\mathit{Service}}[X]$ in the property corresponds to the passivation of the service *Service*. The passivation is performed at computer $\mathit{Computer}_2$.

In the previous example the services themselves decide when to migrate. Some systems may perform forced migration where a monitoring component migrates the services as reaction to events received. The event themselves may include information of which service to migrate.

Consider the following property,

$$
\begin{array}{l}
\textbf{property}\ \mathit{forced\_migration}(\mathit{Callback})\ \{ \\
\quad [\overleftrightarrow{\mathit{Callback}}(\mathit{System})].(\mathbf{E}(\langle\overleftrightarrow{\mathit{Service}}[X], \{\mathit{Computer}_1\}, \{\mathit{Computer}_2\}\rangle)) \ \wedge \\
\quad [-].\mathit{forced\_migration}(\mathit{Callback}) \\
\}
\end{array}
$$

Callback is the channel at which the process performing the migrations receives the events. For simplicity, we assume the system to be migrated is received as the event itself. Property *forced_migration* holds when systems are migrated from one computer to another one after the event triggering the migration is received. $\mathit{Computer}_1$ is the computer where service *Service* is running, $\mathit{Computer}_2$ is the computer to which the service is migrated.

In some systems a number of servers may be dynamically adjusted according to the number of client requests. For example a web server may spawn web server processes when the number of http requests increases. When the number of http requests decreases, a number of servers may be killed. A property specification in this case is:

$$
\begin{array}{l}
\textbf{property}\ \mathit{reduce\_servers}(\mathit{Callback})\ \{ \\
\quad [\overleftrightarrow{\mathit{Callback}}(\mathit{WebServer})].(\mathbf{E}(\langle\overleftrightarrow{\mathit{stop}}(\mathit{WebServer})\rangle.\mathbf{E}(\langle\overleftrightarrow{\mathit{WebServer}}[X]\rangle))) \ \wedge \\
\quad [-].\mathit{reduce\_servers}(\mathit{Callback}) \\
\}
\end{array}
$$

Callback is the channel where the process in charge of killing web servers is waiting for the events. Again, for simplicity we assume the event itself is the web server to kill.

Passivation is not restricted to stopping or changing the location of a process. Passivation can also be used to alter the process executing within a kell. For example, consider the following process specification where the features provided by a process in a kell K are adjusted after an event e is received:

$$\textbf{process}\ \mathit{adjust\_features}()\ \{\ \mathit{callback}(e)\ \triangleright\ (K[X] \triangleright K[P])\ \}$$

In the example, the process for kell K is changed to P when an event e is received in the *callback* channel. Such an specification may be useful for applications executing in handheld devices. When running low on battery, one may be interested in having the applications in the handheld adapt by providing a reduced set of features. Assuming a *low-battery* event, P in the previous specification corresponds to the reduced features of application K available when the device is running at low battery. A kμ property can be specified requiring the passivation after such events are received:

$$\begin{array}{l}\textbf{property}\ \mathit{change\_features}()\ \{\\ \quad ([\overleftrightarrow{\mathit{callback}}(\mathit{Event})].\mathbf{E}(\langle \overleftrightarrow{K}[X]\rangle))\ \wedge\ [-].\mathit{change\_features}()\\ \}\end{array}$$

Event in the property corresponds to the *low-battery* event.

In Appendix C, we list the complete specification for the core API along with the properties specified in this section. The listing can be fed to the tools described in Chapter 6.

We conclude this section by observing that because multiple DEBSs can be modelled at a given time, it is possible to specify that properties are met on systems composed of multiple DEBSs. Such properties can be of interest when integrating multiple DEBSs.

5.2.6 Optional API

Knowing the kinds of events that will be published allows DEBSs to optimize the routing of events to subscribers. Hence, several DEBSs require publishers to advertise events prior to publication (e.g., Hermes [134] and Siena [38]; details in Section 2.2). For these systems, the core API is extended with event advertisement and the ability to update the time-to-live of both subscriptions and advertisements. Specifically, the Optional API extends the Core API with the following calls:

```
advertise(filter, ttl) → advertisement
unadvertise(advertisement)
renew_sub(subscription, ttl)
renew_adv(advertisement, ttl)
```

```
process subscription(notify, renew_subsc, filter, callback, ttl) {
  var vttl := ttl in (
    notify(deliver, event) ◇ (
      if (@filter(event) and @ttl()) then
        @deliver(callback, event)
      fi
    )
    |
    renew_subsc(nttl) ◇ (
      vttl := nttl
    )
  )
}
```

Figure 5.6: Subscription Process for Optional API

A `filter` in an advertisement is an expression determining the events being advertised; `ttl` determines how long should the advertisement be kept in the system.

Based on the specification for the Optional API [133], when an event is published there is no way to guarantee the announcer of the event is the same component which advertised the event. In practice, DEBSs identify the component either explicitly because a parameter is passed to the call identifying the component ([e.g., 38, 134]), or implicitly because the API implementation attaches identifying information to the requests when communicating with the system ([e.g., 119]). In any case, to simplify the representation of advertisements we alter the `publish` call in the API by adding as parameter the advertisement. Notice without this extra parameter we would need to keep a list of advertisements and, upon event publication, search in the list for a matching advertisement.

Compared to the Core API, support for advertisements in the Optional API is the most relevant feature addition. Therefore, we further simplify our representation of the Optional API by excluding time-to-live renewal calls. The calls require the ability to update time-to-live values. For subscriptions, a possible specification is shown in Figure 5.6.

A kell-m model representing generic functionality of a DEBS supporting the Optional API with our simplifications is shown in Figure 5.7. The *subscription* process is the same as the one for *coreapi_debs* (Figure 5.2). The `advertise` call in the API is modelled as channel *advertise* receiving the filter, time-to-live, and a return channel. The return channel is used to return a newly created advertisement a.

When an event is published and before notifying subscribers, a process at a checks if

```
process optapi_debs(subscribe, unsubscribe, advertise, unadvertise, publish, deliver) {
  fresh sem (
    sem() |
    var subsc := [ ] in (
      subscribe(filter, callback, ttl, rc) ◇ (
        fresh s(
          subscription(s, filter, callback, ttl)
          |
          sem() ▷ (subsc :=_s @cons(s, *subsc)) ▷ (rc(s) | sem())
        )
      )
      |
      advertise(filter, ttl, rc) ◇ (
        fresh a(
          rc(a) | advertisement(a, filter, ttl)
        )
      )
      |
      publish(a, e) ◇ (
        @a(subsc, deliver, e)
      )
      |
      unsubscribe(s) ◇ (
        sem() ▷ (subsc :=_s @del(*subsc, s)) ▷ sem()
      )
      |
      unadvertise(a) ◇ 0
    )
  )
}
```

Figure 5.7: Optional API Specification

```
process advertisement(advnotify, filter, ttl) {
   advnotify(subsc, deliver, event) ◇(
      if (@filter(event) and @ttl()) then
         foreach s in *subsc do @s(deliver, event) done
      fi
   )
}
```

Figure 5.8: Advertisement Process for Optional API

the event being published is accepted by the advertised filter. The actual notification is specified in the advertisement process depicted in Figure 5.8.

Safety and liveness properties specified for the Core API in Section 5.2.2 apply to the Optional API as well. Safety properties *c_of_interest_only* and *c_delivery_before_subsc* can be used without modification. The property *c_delivery_before_pub* needs to be modified to include the advertisement when events are published:

property *o_delivery_before_pub*() {
$\langle \overleftrightarrow{deliver}(Callback,\ Event) \rangle \ \vee$
$\langle -\{\overleftrightarrow{publish}(Adv,\ Event), \overleftrightarrow{subscribe}(Filter,\ Callback,\ Ttl,\ Rc\}) \rangle .o\_delivery\_before\_pub()$
}

The following safety property, exclusive to the Optional API, requires published events to match their advertisements:

property *o_matches_adv*() {
$\mathbf{E}_e(\langle \overleftrightarrow{publish}(Adv,\ E) \rangle) \ \Rightarrow$
$\mathbf{E}_e(\langle \overleftrightarrow{advertise}(Filter,\ Ttl,\ Rc) \rangle .\mathbf{E}_e(\langle \overleftrightarrow{Rc}(Adv) \rangle .c\_interested(Filter,\ E)))$
}

Liveness property *c_all_notified* needs to be modified to take into consideration the possibility of unadvertisements. First we specify an auxiliary property holding when an advertisement is created and an event of interest to a subscriber is published using the advertisement:

property *o_adv_sub_pub*(*FilterA, TtlA, Filter, Callback, Ttl*) {
$\mathbf{E}_e(\langle \overleftrightarrow{advertise}(FilterA,\ TtlA,\ RcA) \rangle .\mathbf{E}_e(\langle \overleftrightarrow{RcA}(Adv) \rangle)) \ \wedge$
$\mathbf{E}_e(\langle \overleftrightarrow{subscribe}(Filter,\ Callback,\ Ttl,\ SubRc) \rangle .\mathbf{E}_e(\langle \overleftrightarrow{SubRc}(S) \rangle)) \ \wedge$
$\mathbf{E}_e(\langle \overleftrightarrow{publish}(Adv,\ Event) \rangle .(c\_interested(Filter,\ Event) \wedge c\_interested(FilterA,\ Event)))$
}

FilterA and *TtlA* are the advertisement's filter and time-to-live. *Filter*, *Callback*, and *Ttl* are the parameters of the subscription. The liveness property can now be specified as:

$$
\begin{aligned}
&\textbf{property } \mathit{o\_all\_notified}() \ \{ \\
&\quad \mathit{o\_adv\_sub\_pub}(\mathit{FilterA},\ \mathit{TtlA},\ \mathit{Filter},\ \mathit{Callback},\ \mathit{Ttl}) \Rightarrow \\
&\quad (\mathbf{E}_e(\langle \overleftrightarrow{\mathit{deliver}}(\mathit{Callback},\ \mathit{Event})\rangle) \ \vee \\
&\quad\ \mathbf{E}_e(\langle \overleftrightarrow{\mathit{unsubscribe}}(S)\rangle) \ \vee \\
&\quad\ \mathbf{E}_e(\langle \overleftrightarrow{\mathit{unadvertise}}(\mathit{Adv})\rangle) \ \vee \\
&\quad\ \mathbf{E}_e(\langle \overleftrightarrow{\mathit{Ttl}}(\mathit{TtlRc})\rangle.\neg \mathit{returns\_true}(\mathit{TtlRc})) \ \vee \\
&\quad\ \mathbf{E}_e(\langle \overleftrightarrow{\mathit{TtlA}}(\mathit{TtlRA})\rangle.\neg \mathit{returns\_true}(\mathit{TtlRcA})))
\end{aligned}
$$

The kell containment and kell passivation properties specified for the Core API also apply to the Optional API. Properties where actions on the *publish* channel are specified (e.g., *c_from_site_only*, *c_not_at_site*) need to be adjusted to include the advertisement.

5.3 Rebeca

Most DEBSs readily support the Common API, or can be easily modified to support it [133]. An example is Rebeca, also known as the Rebeca Event-Based Electronic Commerce Architecture. Developed by Gero Mühl [119], Rebeca is a DEBS originally proposed for the study of event routing algorithms [118, 120, 119, 121]. Rebeca provides content-based event subscriptions with event replaying capabilities (cf. Sections 2.9 and 5.2.1).

The calls in the API provided by Rebeca are:

```
void publish(Event e)
void subscribe(Subscription s,EventProcessor proc)
void unsubscribe(Subscription s)
void advertise(Advertisement a)
void unadvertise(Advertisement a)
```

Subscriptions and advertisements are instances of classes `Subscription` and `Advertisement`. These classes are subclasses of a `Filter` class. Each filter instance implements a `match` method, able to identify if a given event is of interest. Parameter `proc` in the `subscribe` call is an instance of `EventProcessor`, and provides a callback `process` method to be invoked when an event is notified.

Comparing Rebeca's API with the Optional API presented in the previous section, subscriptions and advertisements in Rebeca correspond to filters in process *optapi_debs*,

and event processors correspond to callbacks. Since REBECA does not have time-to-live support, REBECA's subscription call can be modelled with *optapi_debs*'s *subscribe* call, specifying *true* as time-to-live. Recall from Section 3.6.5, a process at *true*, always returns two other channels, t and f, and writes on the first one.

Later releases of REBECA support scopes, a hierarchical structuring mechanism for DEBSs [68, 69, 67]. The specification of scopes in this section showcases the use of kell-m to represent advanced features proposed for DEBSs.

A scope is a group of publishers, subscribers, and other scopes. Publishers and subscribers are called *simple components*, whilst scopes are called *complex components*. Formally, with $\mathcal{C}$ the set of simple components, $\mathcal{S}$ the set of scope (complex) components, and E a binary relation over $C = \mathcal{C} \cup \mathcal{S}$, the scope hierarchy is modelled by a directed acyclic graph $G = (C, E)$.

To support scopes, REBECA's API is extended with the following calls:

```
void joinScope(Component c, Scope s)
void leaveScope(Component c, Scope s)
```

Class `Scope` is a subclass of `Component`. `joinScope` and `leaveScope` allow a component to join and leave scopes.

Key to the concept of scopes, is the fact that visibility of events is limited to the components enclosed within a scope. Formally, visibility v is a reflexive and symmetric relation over C. Having $super(X) = X' | (X, X') \in E$ denote the set of parents of X in the scope hierarchy, two components X and Y are visible to each other, $v(X, Y)$, if they both are part of a common superscope:

$$v(X, Y) \Leftrightarrow X = Y \vee v(Y, X) \vee v(X', Y) \text{ with } X' \in super(X)$$

When a component publishes an event, the event is delivered to all subscribed components visible to the publisher component.

For example, consider the scope hierarchy depicted in Figure 5.9. Boxes are used to represent scopes, and circles to represent simple components. Dashed arrows represent the propagation of an event published by C_4. Propagation occurs only to visible components: S_3, S_2, C_2 and C_3. If interested, these are the only components notified of such an event. An event published by C_2 is visible to all components, and an event notified by C_1 is only visible to S_1 and C_2.

5.3.1 Specification of Scopes and Event Visibility

Because a component in REBECA can belong to more than one scope, kell containment cannot be used to model scope-containment relationships. Hence, we model the scope hier-

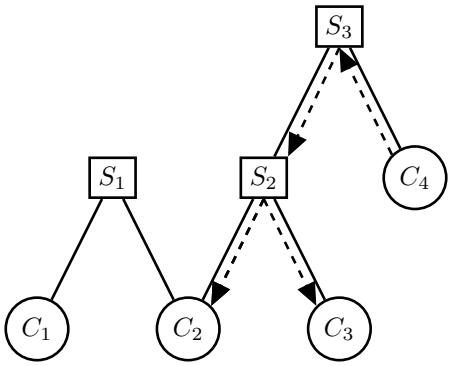


Figure 5.9: Sample Scope Hierarchy

archy as a list of pairs $[S, C]$, indicating that component C is in scope S. The representation of the scope hierarchy for Figure 5.9 is $[[S_1, C_1], [S_1, C_2], [S_2, C_2], [S_2, C_3], [S_3, S_2], [S_3, C_4]]$.

Given a component and the scope hierarchy *shchy*, the following process at channel *super* returns the list of parent superscopes for the component:

$$
\begin{array}{l}
\mathit{super}(c,\ \mathit{shchy},\ \mathit{accum},\ \mathit{rc})\ \diamond\ (\\
\quad \textbf{match}\ \mathit{shchy}\ \textbf{with} \\
\qquad\qquad [\,]\ \triangleright\ \overline{\mathit{rc}}(\mathit{accum}) \\
\qquad \textbf{or}\ s :: ss\ \triangleright\ \textbf{if}\ (@\mathit{member}(c, @\mathit{pos}(s,\ 2)))\ \textbf{then} \\
\qquad\qquad\qquad \overline{\mathit{super}}(c, ss, @\mathit{cons}(@\mathit{pos}(s,\ 1), \mathit{accum}), \mathit{rc}) \\
\qquad\qquad\quad \textbf{else} \\
\qquad\qquad\qquad \overline{\mathit{super}}(c,\ ss,\ \mathit{accum},\ \mathit{rc}) \\
\qquad\qquad\quad \textbf{fi} \\
)
\end{array}
$$

Partial results are stored in parameter *accum*. For example, when $\overline{\mathit{super}}(C_2, \mathit{shchy}, [\,], \mathit{rc})$, the list returned in channel *rc* is $[S_1, S_2]$.

The visibility v, between two components c_1 and c_2, is determined by the following process at channel v:

$$
\begin{array}{l}
v(c_1, c_2, \mathit{shchy}, \mathit{rec}, \mathit{rc})\ \diamond\ (\\
\quad \textbf{if}\ (c_1 = c_2\ \textbf{or}\ (\mathit{rec} = 0\ \textbf{and}\ @v(c_2, c_1, \mathit{shchy}, 1, \mathit{rc})))\ \textbf{then} \\
\qquad \textbf{fresh}\ t, f\ (\overline{\mathit{rc}}(t,\ f) \mid \overline{t}()) \\
\quad \textbf{else} \\
\qquad \overline{\mathit{vsuper}}(@\mathit{super}(c_1, \mathit{shchy}, [\,]), c_2, \mathit{shchy}, \mathit{rc}) \\
\quad \textbf{fi} \\
)
\end{array}
$$

Notice the process at channel v closely follows the definition of visibility $v(X, Y)$ previously presented. Parameter *rec* is used to flag the invocation where components c_1 and c_2 have been swapped. Otherwise, the recursion may never end. At channel *vsuper*, the following process checks for visibility between a component c and a list of scopes *supscopes*. Parameter *shchy* represents the scope hierarchy:

```
vsuper(supscopes, c, shchy, rc) ◇ (
   match supscopes with
            [ ] ▷ fresh t, f (rc(t,f) | f())
      or s :: ss ▷ if (@v(s, c, shchy, 0)) then
                      fresh t, f (rc(t,f) | t())
                   else
                      vsuper(ss, c, shchy, rc)
                   fi
)
```

Using the previous processes, the specification for a DEBS with support for scopes is depicted in Figure 5.10. Because of the need to know the components involved in the interactions, most of the methods are extended with a parameter representing the component. Since REBECA does not support time-to-live, this feature is excluded from the specification to simplify the model.

Besides *subsc*, the list of subscribers used in the to previous models presented in this section, a list *shchy* storing the scope hierarchy is now used. Two semaphores are used as well, *sem_s* is used to guarantee exclusive access to *subsc*, and *sem_h* for *shchy*. The scope hierarchy is maintained by the processes at channels *joinscope* and *leavescope*.

The actual check for visibility is modelled in the *subscription* process as shown in Figure 5.11. Notice the extra parameter passed to the *notify* channel in the *subscription* process. It corresponds to the publishing component. An event is delivered if it is of interest and if the subscriber and publisher components are in a common scope.

The advertisement module, as shown in Figure 5.12, receives the component for which the advertisement was created, and passes that information as a parameter when communicating with subscription processes.

According to Fiege [67], structuring DEBS applications using scopes allows the examination of the following issues:

- *Heterogeneity*: scopes can serve as bridges between different event models. In this use, scopes serve as mediators for the simple components. For example, an event e in a scope S_1 can be represented as another type of event in another scope S_2. When an

```
process scoped_debs(subscribe, unsubscribe, advertise, unadvertise, publish,
                    joinscope, leavescope, deliver) {
  fresh sem_s, sem_h (
    sem_s() | sem_h() |
    var subsc := [ ], shchy := [ ] in (
      subscribe(component, filter, callback, rc) ◇ (
        fresh s(
          subscription(s, component, filter, callback)
          |
          sem_s() ▷ (subsc :=_s @cons(s, *subsc)) ▷ (rc(s) | sem_s())
        )
      )
      |
      advertise(component, filter, rc) ◇ (
        fresh a(
          rc(a) | advertisement(a, component, filter)
        )
      )
      |
      publish(a, e) ◇ (
        @a(subsc, shchy, deliver, e)
      )
      |
      unsubscribe(s) ◇ (
        sem_s() ▷ (subsc :=_s @del(*subsc, s)) ▷ sem_s()
      )
      |
      unadvertise(a) ◇ 0
      |
      joinscope(component, scope) ◇ (
        sem_h() ▷ (shchy :=_s @cons([scope, component], *shchy)) ▷ sem_h()
      )
      |
      leavescope(component, scope) ◇ (
        sem_h() ▷ (shchy :=_s @del(*shchy, [scope, component])) ▷ sem_h()
      )
    )
  )
}
```

Figure 5.10: Scoped DEBS Specification

```
process subscription(notify, component, filter, callback) {
    notify(deliver, pubcomponent, event) ◇ (
        if (@filter(event) and @v(component, pubcomponent)) then
            @deliver(callback, event)
        fi
    )
}
```

Figure 5.11: Subscription Process for Scoped DEBSs

```
process advertisement(advnotify, pubcomponent, filter) {
    advnotify(subsc, shchy, deliver, event) ◇ (
        if (@filter(event)) then
            foreach s in *subsc do @s(deliver, pubcomponent, event) done
        fi
    )
}
```

Figure 5.12: Advertisement Process for Scoped DEBSs

event e is generated by a component in S_1, the event can be received by S_1, assuming S_1 is subscribed. S_1 then transforms the event into the representation used in S_2. Once the event has been transformed, S_1 republishes the event in the S_2 scope.

- *Flexible Configuration*: scopes can be adapted to support application specific requirements. For example a scope can provide event ordering guarantees for events generated within the scope, or implement specific delivery semantics.

- *Security and Session Support*: security policies can be localized within scopes. Dynamic scopes can be created for a duration of a session. All event interchanges between components in the session are then managed by the same session scope. This use of scopes requires knowledge of the components that will be participating in a session before the session starts.

5.3.2 Safety and Liveness Properties for Scoped DEBSs

Safety properties specified for simple DEBSs specified in Section 5.2.2 also apply to scoped DEBSs but require modification to include the new parameters for component and to

exclude time-to-live conditions:

$$\begin{array}{l}\textbf{property } s\_of\_interest\_only() \ \{ \\ \quad \neg \mathbf{E}_e(\langle \overleftrightarrow{subscribe}(Component,\ Filter,\ Callback,\ Rc)\rangle . \\ \qquad (\mathbf{E}_e(\langle \overleftrightarrow{deliver}(Callback,\ Event)\rangle) \wedge \neg c\_interested(Filter,\ Event)) \\ \quad) \\ \}\end{array}$$

$$\begin{array}{l}\textbf{property } s\_delivery\_before\_subsc() \ \{ \\ \quad \langle \overleftrightarrow{deliver}(Callback,\ Event)\rangle \ \vee \\ \quad \langle -\overleftrightarrow{subscribe}(Component,\ Filter,\ Callback,\ Rc)\rangle . s\_delivery\_before\_subsc() \\ \}\end{array}$$

An extra property, specific to scoped DEBSs, requires delivery of events to visible components only:

$$\begin{array}{l}\textbf{property } s\_scoped\_delivery() \ \{ \\ \quad \mathbf{E}_e(\langle \overleftrightarrow{deliver}(Callback,\ Event)\rangle) \ \Rightarrow \ (\\ \qquad \mathbf{E}_e(\langle \overleftrightarrow{subscribe}(Component,\ Filter,\ Callback,\ Rc)\rangle) \ \wedge \\ \qquad \mathbf{E}_e(\langle \overleftrightarrow{advertise}(PubComponent,\ FilterA,\ RcA)\rangle . \\ \qquad\quad \mathbf{E}_e(\langle \overleftrightarrow{RcA}(Adv)\rangle . \\ \qquad\qquad \mathbf{E}_e(\langle \overleftrightarrow{publish}(Adv,\ Event)\rangle . s\_visible(Component,\ PubComponent)) \\ \qquad\quad) \\ \qquad) \\ \quad) \\ \}\end{array}$$

Property *s_visible* is specified as:

$$\textbf{property } s\_visible(C_1, C_2) \ \{ \ \langle \overleftrightarrow{v}(C_1, C_2, Shchy, N, Rc)\rangle . returns\_true(Rc) \ \}$$

The liveness property for scoped DEBSs requires notification of all visible and interested components. We start by specifying an auxiliary property holding when an advertisement is created and a subscribed and interested component is visible:

$$\begin{array}{l}\textbf{property } s\_adv\_sub\_pub\_visible(Component,\ PubComponent,\ FilterA,\ Filter,\ Callback) \ \{ \\ \quad \mathbf{E}_e(\langle \overleftrightarrow{advertise}(PubComponent,\ FilterA,\ RcA)\rangle . \mathbf{E}_e(\langle \overleftrightarrow{RcA}(Adv)\rangle)) \ \wedge \\ \quad \mathbf{E}_e(\langle \overleftrightarrow{subscribe}(Component,\ Filter,\ Callback,\ SubRc)\rangle . \mathbf{E}_e(\langle \overleftrightarrow{SubRc}(S)\rangle)) \ \wedge \\ \quad \mathbf{E}_e(\langle \overleftrightarrow{publish}(Adv,\ Event)\rangle . (c\_interested(Filter,\ Event) \wedge c\_interested(FilterA,\ Event) \wedge \\ \qquad\qquad\qquad\qquad s\_visible(Component,\ PubComponent))) \\ \}\end{array}$$

An event should be delivered unless the component unsubscribes, the publisher unadvertises, or the publisher and subscriber no longer see each other:

property *s_all_notified*() {
s_adv_sub_pub_visible(*Component, PubComponent, FilterA, Filter, Callback*) ⇒
($\mathbf{E}_e(\langle \overleftrightarrow{deliver}(Callback,\ Event)\rangle)$ ∨
$\mathbf{E}_e(\langle \overleftrightarrow{unsubscribe}(S)\rangle)$ ∨
$\mathbf{E}_e(\langle \overleftrightarrow{unadvertise}(Adv)\rangle)$ ∨
¬*s_visible*(*Component, PubComponent*))

Kell containment and kell passivation properties specified in Section 5.2.4 apply to scoped DEBSs and do not require adjusting. The unordered delivery property just needs to be modified to include scope-specific parameters:

property *s_ordered_delivery*() {
$\neg(\mathbf{E}_e(\langle \overleftrightarrow{publish}(Adv_1, E_1)\rangle).\mathbf{E}_e(\langle \overleftrightarrow{publish}(Adv_2, E_2)\rangle)) \Rightarrow$
$\mathbf{E}_e(\langle \overleftrightarrow{deliver}(C_1, E_1)\rangle).\mathbf{E}_e(\langle \overleftrightarrow{deliver}(C_2, E_2)\rangle))))$
}

5.4 NaradaBrokering

With the exception of properties imposing locality constraints via kell containment and kell passivation conditions (cf. Section 5.2.4), the use of kells in the models presented so far is concealed in the sugared constructs of hl-kell-m used in the specifications (e.g., variables, control structures, list support). In this section we show how kells can be explicitly used to model structural aspects of DEBSs. In particular we look at NaradaBrokering, a DEBS where components are organized in a hierarchical structure [130].

Besides publishers and subscribers, in NaradaBrokering there is a special group of components called *brokers*. Brokers are in charge of routing the events across the system from publishers to subscribers. Brokers are grouped in clusters, clusters are grouped in super-clusters, and super-clusters are grouped in super-super-clusters. By default the cluster containment hierarchy has four levels. Using NaradaBrokering's terminology, brokers are at leaf level, clusters are at level 0, super-clusters at level 1, and so on. A broker can only be in one cluster. The term *broker network* is used to refer to the complete broker hierarchy.

At least one broker is required when NaradaBrokering is started. Brokers in the network provide a `join(level)` operation that can be invoked by other brokers wanting to join the

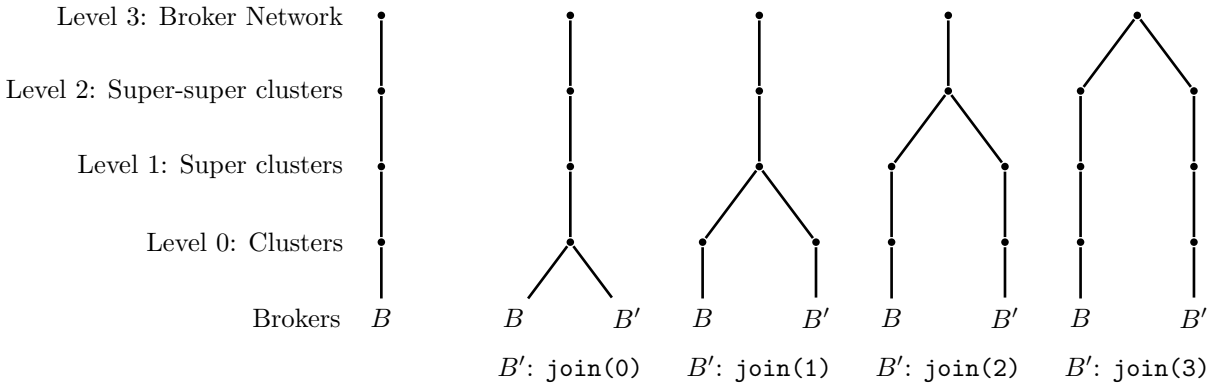


Figure 5.13: NaradaBrokering Broker Network Creation

network. As shown in Figure 5.13, when `level` is 0, the requesting broker (B') joins the same cluster where the contacted broker (B) is located. When `level` is 1, a new cluster is created and the requesting broker becomes the only member of the cluster. The new cluster is located in the same super-cluster as the cluster where the contacted broker is located. If `level` is 2, a cluster and super-cluster are created. Assuming the default four levels in the cluster containment hierarchy, the maximum allowed value for `level` is 3.

Brokers running on a computer register a network port number at the computer. Once a broker has registered, other brokers can obtain a communication link with the broker by knowing the computer where the broker runs and the port where the broker is waiting for requests. This setup is typical in applications using network sockets for communication. We model this operation with the *host* process specified in Figure 5.14.

A list of brokers is kept at each host. Elements in the list are pairs (p, j) with p a port number and j a join channel. Join channels are the channels at which brokers wait for requests from other brokers wanting to join the broker network. As previously mentioned, ports correspond to a destination network port on the host being modelled. When the host receives a connection request on a particular port, it looks for the join channel of the broker associated to the port.

5.4.1 Representation of the Broker Network

We model NaradaBrokering broker networks using kells. Specifically, brokers, clusters, super-clusters and so on execute within kells that match the structure of the broker network. A broker network is setup by a process at channel *brokernw* in the process *narada* depicted in Figure 5.15. A process at channel *cluster* is used to represent the functionality of clusters and all groupings higher up in the hierarchy.

```
process host(addbroker, connect) {
  fresh sem (
  sem() |
    var brokers := [ ] in (
      addbroker(joinc, port) ◇ (
        sem() ▷ (brokers :=_s @cons([port, joinc], *brokers)) ▷ sem()
      )
      |
      connect(port, rc) ◇ (
        match *brokers with
                [ ] ▷ rc(null)
          or p :: ps ▷ if (@pos(p, 1) = port) then
                         rc(@pos(p, 2))
                       else
                         @connect(ps, rc)
                       fi
      )
    )
  )
}
```

Figure 5.14: Host Representation for NaradaBrokering

Given a number of clustering levels, and a channel for communication with the parent cluster for coordination of join operations, the process at channel *brokernw* returns a process with as many nested kells as one less than the number of levels specified. Inside each kell there is a cluster process as returned by the process at *cluster*. Notice two return channels are used at *brokernw*. *rj* is used to return a communication channel where the cluster at level 0 can be contacted for join requests; *rc* is used to return the actual process modelling the broker network. As done in NaradaBrokering, a join request is first received by a broker which in turn forwards the request to its cluster which, if necessary (i.e., $level > 0$), forwards the request up the broker network until it is received by the process in charge of the requested join level.

At each level, a process (as returned by the process at channel *cluster*) waits for join requests at channel *join*. The first argument in the communication at *join* is the name of the kell where the broker wishing to join is executing. If the request is for level 0, the broker is transported to the cluster using kell passivation: $K[X] \triangleright K[X]$. When a broker joins the broker network it receives as return value the join channel of the cluster where it

```
process narada(brokernw, cluster) {
  brokernw(levels, pjoin, rj, rc) ◇ (
    if (levels >= 0) then
      fresh K, join (
        @cluster(join, levels, pjoin)(pcluster) ▷ (
          @brokernw(levels − 1, join, rj)(sclusters) ▷ rc(K[pcluster|sclusters])
        )
      )
    else
      rc(zero) | rj(pjoin)
    fi
  )
  |
  cluster(join, level, pjoin, crc) ◇ (
    crc(
      join(K, blevel, rc) ◇ (
        if (level = 0) then
          K[X] ▷ K[X] |
          rc(join)
        elsif (blevel > level) then
          pjoin(K, blevel, rc)
        else
          fresh SK, njoin (
            @brokernw(level − 1, njoin, rj)(sclusters) ▷ (
              SK[sclusters] | rj(sjoin) ▷ sjoin(K, 0, rc)
            )
          )
        fi
      )
    )
  )
}
```

Figure 5.15: Model for NaradaBrokering Broker Network

has been placed, $\overline{rc}(join)$.

If the requested level is higher than the level of the cluster, the request is handed to the parent of the cluster in the cluster containment hierarchy. Notice the parent's own join

operation is available at the *pjoin* channel. If the requested level is not 0 and matches the cluster's own level, new cluster groupings are created as previously illustrated in Figure 5.13, and the join request is handed to the new cluster at level 0 in the newly created branch.

A broker is modelled using the following process:

$$
\begin{array}{l}
\textbf{process}\ \mathit{broker}(\mathit{bjoin},\ \mathit{cjoin})\ \{ \\
\quad \mathit{bjoin}(\mathit{kell},\ \mathit{level},\ \mathit{rc})\ \diamond\ \mathit{cjoin}(\mathit{kell},\ \mathit{level},\ \mathit{rc}) \\
\}
\end{array}
$$

The process receives the join channel where the broker will wait for join requests, and the join channel for the cluster where the broker has been placed.

Initially, a broker network of four levels and a single broker is setup by the following process:

$$
\begin{array}{l}
\mathit{narada}(\mathit{brokernw},\ \mathit{cluster})\ |\ \mathit{host}(\mathit{addbroker},\ \mathit{connect})\ | \\
\textbf{fresh}\ \mathit{rj}\ (\\
\quad @\mathit{brokernw}(3, \mathit{null}, \mathit{rj})()\ \triangleright\ \ (\\
\quad\quad \mathit{rj}(\mathit{cjoin})\ \triangleright\ (\\
\quad\quad\quad \textbf{fresh}\ \mathit{bjoin}, B_1\ (\\
\quad\quad\quad\quad B_1[@\mathit{cjoin}(B_1, 0)(\mathit{cj}) \triangleright (\mathit{broker}(\mathit{bjoin},\ \mathit{cjoin})\ |\ \overline{\mathit{addbroker}}(\mathit{bjoin}, 1025) \cdots)] \\
\quad\quad\quad) \\
\quad\quad) \\
\quad) \\
)
\end{array}
$$

The join channel for the only cluster at level 0 is *cjoin*. A new channel *bjoin* and kell B_1 are created. Within kell B_1 the only broker sets its join operation using *broker*(*bjoin, cjoin*) and registers itself as a broker on its host at port 1025. If a second broker knows there is a broker on the host at port 1025, it can join the broker network as follows:

$$
\begin{array}{l}
\textbf{fresh}\ \mathit{bjoin}, B_2(\\
\quad B_2[@\mathit{connect}(1025)(\mathit{joinc}) \triangleright @\mathit{joinc}(B_2, 0)(\mathit{cjoin}) \triangleright (\\
\quad\quad \mathit{broker}(\mathit{bjoin}, \mathit{cjoin}) | \overline{\mathit{addbroker}}(\mathit{bjoin}, 7070)) \\
\quad] \\
)
\end{array}
$$

In the example the new broker registers itself at port 7070. Because the broker requested a level 0 join, no new clusters are created.

5.4.2 Safety Property for Broker Network

Since the structure of the broker network is specified in model *narada* using kell containment, when a broker joins a cluster, the process modelling the cluster must be within *levels* of other kells, where *levels* is the number of levels in the clustering containment hierarchy (four for the broker networks depicted in Figure 5.13):

$$
\begin{array}{l}
\textbf{property}\ \mathit{cluster\_nesting}(\mathit{levels})\ \{ \\
\quad \mathbf{E}_e(\langle \overleftrightarrow{\mathit{cluster}}(\mathit{join},\ 0,\ \mathit{pjoin},\ \mathit{crc})\rangle. \\
\qquad \mathbf{E}_e(\langle \overleftrightarrow{\mathit{join}}(\mathit{broker\_kell}, 0, \mathit{rc})\rangle. \\
\qquad\quad \mathbf{E}(\langle \overleftrightarrow{\mathit{broker\_kell}}[X], \mathit{Kr}, \mathit{Kw}\rangle.(|\mathit{Kr}| = \mathit{levels})))) \\
\}
\end{array}
$$

Property *cluster_nesting* specifies a communication representing the creation of a cluster process at level zero (the cluster level), followed by a broker join request, always followed by the passivation of the broker's kell. The passivation of the broker's kell occurs when the broker is included into the broker network. The property requires the broker's passivation to be within *levels* kells.

5.4.3 Adaptation of Broker Network

Although not currently supported in NaradaBrokering, the brokering network could adapt by adding brokers as the load of the network increases and by reducing the number of brokers as the load decreases. In Figure 5.16 we show a model based on the *narada* process representing the addition of a broker every time a *"high-load"* event is received.

A variable port is used to keep track of the ports where brokers have already being assigned. A semaphore *sem* is used to guarantee exclusive access to the *port* variable. Once the broker is created (*brokernw*) and the first broker has been setup, a subscription to events *"high-load"* is obtained. We assume a DEBS with no time-to-live support for subscriptions.

Channel *loadcb* is registered as the callback in the event subscription. Upon reception of an event the process at channel *loadcb* creates a new broker, adds the broker to the broker network, and registers the broker at the next available port.

The following property specifies the creation of a broker after each *"high-load"* event is notified:

$$
\begin{array}{l}
\textbf{property}\ \mathit{add\_broker}()\ \{ \\
\quad \mathbf{E}_e(\langle \overleftrightarrow{\mathit{cluster}}(\mathit{join},\ 0,\ \mathit{pjoin},\ \mathit{crc})\rangle.(\\
\qquad \mathbf{E}_e(\langle \overleftrightarrow{\mathit{loadcb}}(e)\rangle)\ \Rightarrow\ \mathbf{E}(\langle \overleftrightarrow{\mathit{join}}(\mathit{newbroker},\ 0,\ \mathit{rc})\rangle) \\
\quad)) \\
\}
\end{array}
$$

```
process narada_adapt {
   narada(brokernw, cluster)
   |
   host(addbroker, connect)
   |
   var port :=_s 1024 in (
      fresh rj @brokernw(3, null, rj)() ▷ (
         sem̄()
         |
         rj(cjoin) ▷ (
            fresh bjoin, B (
               B[@cjoin(B, 0)(cj) ▷ (broker(bjoin, cjoin) | addbroker̄(bjoin, 1024) ···)]
            )
            |
            @subscribe(loadfilter, loadcb)(s) ▷ (
               loadfilter(e, rc) ◇ (
                  fresh t, f (rc̄(t,f) | if (e = "high-load") then t̄() else f̄() end)
               )
               |
               loadcb(e) ◇ (
                  sem() ▷ (
                     port :=_s *port + 1 ▷ (
                        var p := *port in (
                           sem̄() | fresh nbj, NB (
                              NB[@cjoin(NB, 0)(cj) ▷ (
                                 broker(nbj, cjoin) | addbroker̄(nbj, *p)
                              )]
                           )
                        )
                     )
                  )
               )
            )
         )
      )
   )
}
```

Figure 5.16: Adaptation of Broker Network

After a cluster is created, every time an event is received in channel *loadcb*, the *join* channel of the cluster is used to create a new broker at level 0.

For simplicity we assume in this example all brokers are added on the current server, and all broker additions occur at level 0. In a real-world application, a process such as the one described would be running on each server hosting brokers. Also, as the load decreases, brokers should be removed from the network.

5.4.4 Adaptation of Broker Behaviour

We now illustrate how the behaviour of brokers can be adapted when they join the broker network. Specifically, we assume functionality related to the services that all brokers provide within the broker network is injected into each broker upon joining the network.

In Figure 5.15 a broker being added into the broker network is represented by the action $K[X] \triangleright K[X]$, where K is the name of the kell corresponding to the broker being added. Assuming P is the functionality that is added to each joining broker, the only required change in the *narada* model is to replace the previous action with $K[X] \triangleright K[X|P]$. This new action indicates not only a broker being moved into the network via a kell passivation, but also the adaptation of the process within the broker.

Further, if there is at least one action in P that is observable, we can specify a property requiring all brokers to exhibit the observable action from P once they join the broker network. For simplicity we will assume the observable action is a communication on channel a where the abstraction part of the communication is executed by the broker:

$$
\begin{array}{l}
\textbf{property}\ \mathit{adapt\_broker}()\ \{ \\
\quad \mathbf{E}_e(\langle \overleftrightarrow{\mathit{cluster}}(\mathit{join},\ 0,\ \mathit{pjoin},\ \mathit{crc})\rangle. \\
\qquad \mathbf{E}_e(\langle \overleftrightarrow{\mathit{join}}(\mathit{broker\_kell}, 0, \mathit{rc})\rangle. \\
\qquad\quad \mathbf{E}(\langle \overleftrightarrow{\mathit{broker\_kell}}[X]\rangle.\mathbf{E}(\langle \overleftrightarrow{a}(), \supseteq \{\mathit{broker\_kell}\}, *\rangle)) \\
\qquad) \\
\quad) \\
\}
\end{array}
$$

The previous property specifies that, once a cluster at level 0 is created, join requests for the cluster are followed by the observable action. The abstraction part of the action on a must be located within kell *broker_kell* indicating that the abstraction must have executed within the broker. Similar properties can be specified for other observable actions in P.

5.5 Beyond DEBSs

Kell-m provides the ability to represent systems as processes communicating on channels and passivating kells. As illustrated with the case studies in this chapter, when the communication on channels match actions on the modelled system, property specifications in kμ can be used to determine the conditions under which the actions are executed including the locality of the actions and the order in which actions take place.

When kell passivation is used to model systems where processing at a location can be stopped, altered, or migrated to another location, kμ properties can be specified formalizing the conditions causing the adjustment of location or behaviour.

We expect the modelling and reasoning capabilities of kell-m and kμ illustrated throughout this chapter to be applicable to areas beyond those related to DEBSs. In particular in applications where location-awareness is of interest as well as where changes in the system or environment require the adaptation of the behaviour exhibited by system components.

5.6 Related Work

In this section we first describe representative models proposed for general implicit invocation systems (IISs) and discuss their applicability to DEBSs [55, 63]. We then describe models proposed specifically for DEBSs [119, 16, 32].

Besides the specification of the DEBS event model and the behaviour exhibited by the DEBS middleware, kμ can be used to specify the behaviour of components as it pertains to the reaction and generation of events (cf. examples in Sections 5.2.4 and 5.2.5). At the end of this section we review formalisms proposed for the specification of component behaviour.

5.6.1 Model for Synchronous Implicit Invocation

Dingel et al. propose a formal model for the compositional verification of synchronous implicit invocation systems [55]. In their model, a system S consist of a set of methods $M = \{m_1, m_2, ..., m_n\}$ with one of the methods in M being a distinguished dispatcher method *disp*. The dispatcher method *disp* stores and delivers events e from a set of events E. A binding set $B \subseteq E \times M$, associates events with the methods to be triggered when an event is announced. A method m_i is a UNITY program [39], and it is represented as a 4-tuple $m_i = (V_i, E_i, P_i, S_i)$ where V_i is the set of variables m_i accesses; E_i is the set of events m_i announces; P_i is a boolean expression over V_i, describing the initial states of m_i; and S_i is a set of guarded statements of the form $g \xrightarrow{a} x := exp$. Guard g is a boolean

expression over V_i. If g holds in the current state, then action a is executed, and the value of the variable $x \in V_i$ is set to the expression *exp* over V_i. Assuming event e and predicate p, an action a can be any UNITY action, plus the following communication actions for sending and receiving messages:

- $\langle e, p \rangle !$, send event e to dispatcher if p holds
- $\langle v, p \rangle ?$, an event can be received on variable v if p holds
- $\langle e, p \rangle ?$, event e can be received if p holds

Communication is achieved by matching announcing ($\langle e, p \rangle !$) and consuming actions ($\langle v, p' \rangle ?$ or $\langle e, p' \rangle ?$). On communication a silent action τ occurs and, if the input action was $\langle v, p' \rangle ?$, event e is assigned to variable v. The dispatcher decides which methods should receive the event by looking at the binding $B \subseteq E \times M$. It is not clear from [55], if it is possible to alter binding B while the system is in operation, as it is required by the DEBS event model.

An environment method m_E, non-deterministically chooses and executes an action from a set of communication actions $\{a_1, ..., a_n\}$. First-order linear-time temporal logic is used to specify the ongoing behaviour of the system.

In order to define the semantics on an event, the environment is constrained to be a method that just announces the event. The focus is not on issues related to delivery policies and event distribution guarantees. It is not clear if by considering each event in isolation, the behaviour exhibited by the system can be fully specified. This is related to the fact that in the work of Dingel et al. an event cannot cause the announcement of other events. Another issue is the assumption of synchronicity by the subscriber: a subscriber is blocked until an event is published and sent to it.

5.6.2 Implicit Invocation Language

Although not proposed specifically for applications that follow the DEBS event model, in [172], Zhang et al. develop an Implicit Invocation Language IIL, and a set of source transformation tools for generating applications verifiable in the Cadence SMV model checker [108]. Properties to verify are expressed using linear temporal logic. Event bindings in IIL are static and explicitly specified. Recall an event binding determines the components that need to be notified of the occurrence of an event (cf. Section 2.4). Since DEBSs support dynamic event binding, representation of DEBS-specific functionality using IIL is not supported. Specifically, the ability in DEBSs for components to introduce new types of events at run time, dynamically subscribe to and unsubscribe from events, and dynamically advertise and unadvertise events.

5.6.3 Logic of Event Consumption and Publication

Fenkam et al. provide operational semantics for an event based system using what the authors call LECAP: Logic of Event Consumption And Publication [63]. In this model, functional components interested in events are invoked by the system and execute only when an event of interest is published. A component P is specified by using the following syntax:

$$
\begin{aligned}
P ::= & x := val \mid P_1 \ ; \ P_2 \mid \textbf{if } cond \textbf{ then } P_1 \textbf{ else } P_2 \textbf{ fi} \mid \\
& \textbf{while } cond \textbf{ do } P_1 \textbf{ od} \mid P_1 \parallel P_2 \mid \textbf{announce } (e) \mid \\
& \textbf{skip} \mid \textbf{await } cond \textbf{ do } P_1 \textbf{ od}
\end{aligned}
$$

Where P, P_1, and P_2 are components. The functional components triggered by the publication of an event of interest are part of the component that published the event: the transitions of the triggered components are internal transitions of the publisher component. This is the main limitation of this model, with respect to its applicability in DEBSs. There is no way to specify *always running* reactive components: execution of the reactive components occur only while reacting to an event, and within the context of the publishing component.

5.6.4 Traces

Mühl proposes a DEBS model based on trace semantics [119]. A trace of a system is a sequence $\alpha = s_1, op_1, s_2, op_2, s_3, ...$ where s_i is the state of the system in step i, and op_i is the operation taken in step i which results in the system going to state s_{i+1}. A specification $\mathcal{S}$ is a set of traces. A system is correct with respect to $\mathcal{S}$, if the system exhibits only traces that are in $\mathcal{S}$.

The sets of traces that a DEBS must exhibit are defined by using predicates for the DEBS operations, quantifiers $\exists$, $\forall$, logical operators $\wedge$ $\vee$ $\Rightarrow$ $\neg$, and temporal operators $\Box$ (always), $\Diamond$ (eventually), and $\circ$ (next).

The model proposed by Mühl assumes instantaneous notification of events, no communication failures, and requires components to be active (connected to the system) in order to be notified of events.

Another work based on trace semantics is by Baldoni et al. [16]. In this work, and in contrast to [119], notifications are not assumed to run instantaneously. Instead of a global trace, a local trace is kept for each component in the system. Each operation in a local trace is tagged with the time, from a global clock, at which the operation is executed. The global trace H for the DEBS is defined as the collection of local traces $\langle$ h_1, h_2, h_3,

... $\rangle$. A subscription interval $I(X, F)$, for a component X and a filter F, is defined as the sequence of all time-tagged operations $\langle Op(X, Y), t\rangle$ in trace h_X executed between subscription $\langle Subs(X, F), s\rangle$ and unsubscription $\langle Unsub(X, F), u\rangle$, with $s \leq t \leq u$. Hence, the time between s and u represents, for the subscriber, the time the subscription was active.

Works based on trace semantics allow the specification of properties based on the order in which the operations should appear in the traces. When a trace for a DEBSs is obtained, one can verify if the trace satisfies the properties. Unless all possible traces are obtained for a given system, one cannot say the system satisfies the properties. Hence, one can only say the properties were satisfied by the system execution from which the trace was obtained.

In contrast, all possible traces for all possible communications can be obtained using kell-m. Traces correspond to paths in the LTS. As shown in Chapter 6, a tool can be developed for the generation of these traces. Moreover, with kell-m and kμ it is possible to specify conditions on the location of the actions as well as the passivation of kells. Such conditions cannot be expressed using only trace semantics.

5.6.5 Other Models

In more recent work [32], Dingel et al. model the behaviour of SIENA [38], a particular DEBS. The purpose of the model is to verify *already developed* DEBS applications. The model itself is specified in BIR, the input language for the model checker Bogor [139]. Two data structures are used in the model: a FIFO communication channel between clients and the DEBS, and an event set to store the events before they are delivered. The event set is used, instead of an event queue, to model the fact that in SIENA there are no order guarantees in the order of delivered events. Only subscribe, unsubscribe and publish operations are supported. No attempts are made in this work to generalize the event model of DEBSs, nor to present a formal specification of the SIENA event model itself.

In contrast, we specify formal models that support, not only the verification of ordering guarantees but also the specification and verification of liveness and safety properties previously identified in the area, the specification and verification of kell containment and kell passivation properties and, in general, application-level properties expressible in terms of communication actions occurring in the modelled systems.

5.6.6 Application Behaviour

Besides the formalisms reviewed above for modelling DEBS (the middleware), other formalisms have also been proposed to model the application level behaviour exhibited by

the publisher and subscriber components. In particular we look at event-condition-action rules and automata-based formalisms.

Event-Condition-Action Rules

As proposed in Rapide [138, 100, 99], and widely used in active databases [167], reactive behaviour can be modelled by ECA rules. An ECA rule specifies input events, possible composite, that must occur for the rule to be triggered. When triggered, a condition on local variables, also part of the rule is then evaluated. Based on the result of the evaluation, an action may be executed. When modelling behaviour using ECA rules, languages must be provided for specifying the input events that trigger the rule, the rule condition, and its actions. The specification of the input events is typically based on event algebras [91, 37], whilst some process calculus formalism may be used to specify the rule actions. In the case of Rapide, specification of temporal conditions in the event part of the rule is supported. Also, it is possible to specify if the events generated by the actions in a rule are independent of each other or not. To be able to decide if two or more components can be composed into a complex component, or to coordinate the interaction of several components, it is necessary to verify that the ECA rules describing the behaviour of each component are composable, and that their actions obtain the target behaviour of the composition.

Interface Automata

Interface automata, proposed by Alfaro and Henzinger [51], have been used to describe the behaviour of reactive systems [159, 162]. Interface automata are specified in an automata-based language. This language is used to capture both, input assumptions about the order in which a component reacts to events, and output guarantees about the order in which the component generates events. Interface compatibility is decided based on an *optimistic* approach. In a traditional, pessimistic approach, two components are compatible if they can be used together in all systems. In the optimistic approach proposed with Interface automata, a helpful environment is assumed: two components are compatible if they can be used together in at least one design. The advantage of the optimistic approach is a simpler model. Interface automata interact through the synchronization of input and output events. Internally, actions of concurrent automata are interleaved asynchronously.

Events are not queued in interface automata: the arrival of a message while in an state not prepared to handle the message, indicates an incompatibility between the environment and the automaton. This is in contrast to DEBSs where events are typically queued until the receiving component is in a state ready to handle the message. Similarly to the DEBS

behaviour, in kell-m a write (i.e., concretion) on a channel is only matched to a read (i.e., abstraction) when both actions are ready for the communication.

Finite State Machines

In [30], Bultan et al. analyze component composition by looking at the conversations between the components. A conversation is the concatenation of all the events exchanged by the components being composed. The behaviour of the components themselves is represented by Mealy machines [109]: finite state machines where output actions can be specified in the transitions. A component is then viewed as a Mealy machine that decides, based on the received events and the events already sent, if a new event should be sent. In contrast to interface automata, Mealy machines interact asynchronously. But in order to perform the analysis of the compositions, it is required to have a global watcher that keeps track of all events as they occur. The authors start by trying to deduce global behaviour by analyzing the behaviour of the components. They find this bottom-up approach flawed and propose to perform a top-down approach instead. Their argument is that given a conversation, it is not possible to find a regular language (*global behaviour*) as its core. Bultan et al. argue this is because of the asynchronous nature of the interactions.

In the top down approach, on the other hand, they start with conversations that represent the intended *global* behaviour of the system, and construct Mealy machines that realize that conversation. Similar to the research we propose, the authors final goal is to understand component composition in distributed systems. Our approach diverges from theirs since our focus is in the specification of functionality in DEBSs, instead of deducing a global (local) behaviour based on a local (global) behaviour.

Statecharts

First proposed by Harel [78], a statechart is a graphical representation used to model reactive systems. Since Harel's original work, multiple variations, both in semantics and structure, have been proposed. UML statecharts [5] is currently the most used variation. Statecharts are, fundamentally, Mealy machines with state entry and exit reactions, hierarchical states, and parallelism. States represent processing stages of the artifact being modelled. Transitions between states are triggered by the reception of events and the evaluation of an optional condition. Actions can be executed as part of the transition. States can be represented by a substatechart or two or more substatecharts operating in parallel.

Harel statecharts assume instantaneous event processing: the reaction to an event occurs in zero time, upon notification of the event. This is not the case when dealing with DEBSs, where events take time to reach subscribed components. Another issue is

the assumption that only one event may happen at a time. UML statecharts do not have this restriction, providing instead a queue of events. Both Harel and UML statecharts assume broadcasting of events, where events are globally visible to all components in the system. In contrast, in DEBSs events are only notified to subscribed components. These differences between the DEBS event model and the statechart event model make the use of statecharts to model behaviour in DEBSs and DEBS applications inadequate. Specifically, event interactions in DEBS cannot be directly specified using statechart event interactions since they are semantically different. This problem is not unique to DEBSs and arises when dealing with any IIS that has an event model incompatible with the statechart event model. The effect of these differences between event models has been the proposal of multiple, different, and sometimes incompatible, statechart variations when using them to model complex IISs ([e.g., 163, 95, 54, 17, 142]).

5.7 Summary and Contributions

DEBSs provide an Application Programming Interface (API) for the interaction with components. Publisher and subscriber components use the calls in the API to announce events and to indicate to the system the events of interest. Although there is no standard DEBS API supported by most DEBSs, there is a proposal for a Common API. The Common API itself is composed of two other APIs: a Core API and an Optional API. The Core API specifies calls for the publication and subscription of events. The Optimal API extends the Core API with calls for the advertisement of events and for the renewal of subscriptions and advertisements.

Since most existing DEBSs can be easily modified to support this Common API, in this chapter we developed models for generic DEBSs that follow these APIs. The models parametrize features typically varying among different DEBS implementations such as filtering and event delivery capabilities. We showed how properties previously identified for DEBSs can be specified in our models. We also showed how the models support the specification of properties beyond the ones previously identified. In particular, we provided examples on how kells can be used in our models to specify application-level properties that deal with locality of publishers and subscribers, as well as kell passivation. These new properties were not expressible using the formalisms previously proposed in this area.

We modelled Rebeca, a particular research DEBS extended with scopes. Scopes are used to structure components in DEBSs. This model showcases the use of kell-m to model an extension to the basic DEBS features.

The previous models represent behaviour exhibited by DEBSs as it pertains to the publication, subscription, and notification of events. To showcase the use of kell-m to model other, possibly implementation-specific, features of interest that may not be exposed

to publishers and subscribers, we modelled the structure of administrative components in NaradaBrokering, a particular DEBS. In NaradaBrokering specialized components, called Brokers, are in charge of the communication within the system. Brokers are grouped in hierarchical clusters. In the model, kells are used to represent such a hierarchy.

Chapter 6

Tools

We illustrate the feasibility of model checking systems represented in kell-m with a prototype tool encoding the kell-m calculus. The tool takes as input system specifications written in hl-kell-m and property formulas written in sugared hl-kμ. Recall hl-kell-m is the syntactically sugared version of kell-m, and hl-kμ is the syntactically sugared version of kμ. Using kell-m's extended labelled transition system semantics the tool supports property verification based on both LTS and reduction semantics.

6.1 Background and Chapter Organization

As discussed in Chapter 3, evolution of kell-m processes can be represented using labelled transition systems (LTSs) and reduction semantics. In Section 3.5.3 we combined both representations into what we called kell-m's extended LTS semantics. The main idea in extended LTS semantics is to expose kell containment and channel information in all LTS transitions, including τ transitions. The motivation for exposing the information is twofold:

- When specifying properties for systems modelled using kell-m, exposing kell containment information allows the specification of kell containment conditions. These conditions have the form *action occurs only within kells* $\mathcal{K}$, *action occurs at least within kells* $\mathcal{K}$, and *action occurs except within kells* $\mathcal{K}$.

- If only τ transitions are considered, the evolution of the process, as modelled by the extended LTS, corresponds to kell-m's reduction semantics. Because in the extended LTS τ transitions include information about the abstraction and concretion being matched, it suffices to observe the τ transitions to determine the channels and kells involved in a communication.

Based on kell-m's extended semantics we have developed a prototype tool for the verification of kell-m processes. The tool fully supports hl-kell-m and hl-kμ, the sugared kell-m and sugared kμ languages presented in Sections 3.3 and 4.3. The tool itself uses two other tools, also developed by us: a checker for kell-m, and a visualizer of kell-m processes.

All input and output, to and from the tools, can be piped to other tools, and be saved into and retrieved from files. Consequently, the tools can be used by themselves or in combination with tools beyond the ones we have developed.

We start our presentation, in Section 6.2, by describing the hl-kell-m checker. The tool receives process and property specifications in hl-kell-m and hl-kμ, and translates them into kell-m and kμ specifications for the kell-m checker. The actual verification of properties is done by the kell-m checker, described in Section 6.3. Related work is presented in Section 6.5, and we conclude the chapter in Section 6.6 with summary and contributions.

We use *italics* for kell-m and kμ expressions, **bold** for reserved words in kell-m and kμ, and `monospace` for Prolog predicates and tool directives. When describing the encoding of process and property expressions in our tool, we combine fonts, e.g., `pred(`P`)`, to indicate an argument P in a Prolog predicate `pred` matches a previously specified process or property expression.

6.2 High-Level kell-m Checker

As illustrated in Figure 6.1, the hl-kell-m checker receives as input process specifications in hl-kell-m, properties in hl-kμ, and verification requests. The tool itself is written in Perl [165].

If no action is specified to the checker it compiles hl-kell-m process specifications into kell-m specifications, and hl-kμ properties into kμ properties. The result of the compilation is stored in a file that can be later provided as input to the kell-m checker or the kell-m visualizer.

Two actions can be optionally specified to the hl-kell-m checker: one action directs the checker to execute the verification requests specified in the input; the other action directs the checker to display, on screen, the kell-m process resulting from the compilation of the provided hl-kell-m processes.

The kell-m viewer receives as input kell-m expressions and outputs latex code that, when processed, produces a document where the kell-m processes are typeset in the same way as the examples of Chapter 3.

Verification requests are given as input to the hl-kell-m checker using the following syntax:

check by *semantics property* **for** *process* **expect** *result*

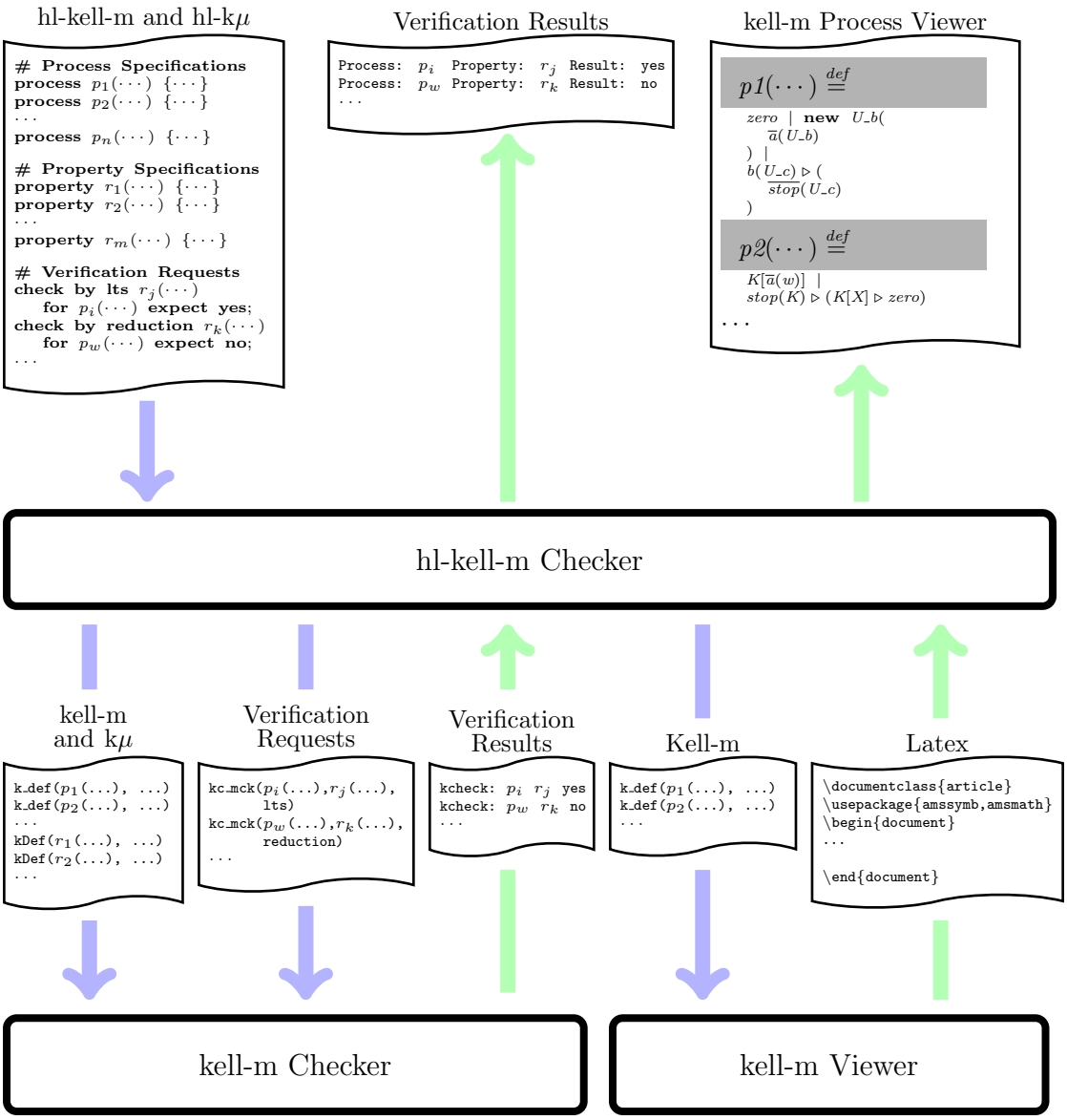


Figure 6.1: High-Level kell-m Checker

semantics can be either **reduction** or **lts**. If **reduction** the LTS used in the verification has τ transitions only (cf. Section 6.3.2). Therefore, properties being verified with reduction semantics can only impose conditions on τ actions. If **lts** semantics are specified, the LTS used in the verification includes all possible transitions.

Notice the verification for both, **lts** and **reduction**, operates on LTSs. If **lts**, the LTS used in the verification is obtained according to the rules in Figure 3.10. If **reduction**, the LTS used in the verification is obtained only with the rules XRestrictTau, XAdvanceTau, XL-React, XL-Suspend, XL-Close, and the symmetrical XR-* rules.

property is the hl-kμ property to verify. It has the form $prop(\widetilde{c})$ and must have been previously defined using:

$$\textbf{property}\ prop(\widetilde{e})\ \{\cdots\}$$

process is the hl-kell-m representation of the system being checked. *process* corresponds to a process invocation $proc(\widetilde{w})$ where *proc* was previously defined using:

$$\textbf{process}\ proc(\widetilde{g})\ \{\cdots\}$$

The **expect** *result* part of the verification request is optional. *result* can be **yes** or **no**. If provided, the tool compares the expected result with the actual result of the verification and, if different, reports the discrepancy. If not provided, the tool just reports the result of the verification.

When the hl-kell-m checker is instructed to perform verification requests, the tool invokes the kell-m checker and passes the processes, property definitions, and list of properties to verify. The results of the execution of the verification requests by the kell-m checker are gathered by the hl-kell-m checker and formatted for the user.

The hl-kell-m checker fully supports the sugared kell-m and kμ described in Chapters 3 and 4. The grammars for these languages are listed in Appendices A.1 and A.2. In the checker `->` and `->>` are used instead of $\triangleright$ and $\diamond$ in trigger definitions. `zero` is used for the null process **0** and `null` is used for the *null* name.

Compilers for hl-kell-m and hl-kμ are generated using Yapp, a Perl extension for the generation of LALR parsers [52]. A full example of the input received by the tool is shown in Appendix C, where the specification of the Core DEBS API (cf. Section 5.2) and its properties are listed.

The checker supports the ability to include process and property definitions from multiple files. A directive:

{`libdir` *directory*}

Allows the specification of search directories where source files may be located. By default these files have the extension `.sk`, for *sugared kell*. The inclusion of a file *file.sk* is done

with the following directive:

{`use` *file*}

The tool searches for *file.sk* in the current directory and all previously specified search directories.

6.3 Kell-m Checker

The core of the hl-kell-m checker tool is the kell-m checker. Written in Prolog, the kell-m checker provides predicates for the verification of kμ properties and for the generation of the extended LTS. The main predicates are:

- `kc_mck(KellProc, Property, TxSemantics)`. This predicate is used to verify if kμ property `Property` holds for kell-m process `KellProc`. `TxSemantics` is an atom indicating whether LTS or reduction semantics should be used in the verification. All three arguments of the predicate are input arguments: given a property, it is not possible to determine all kell-m processes for which the property holds; conversely, given a process it is not possible to determine all properties holding for the process.

- `kdbg_nth(TxSemantics, KellProc, NumIter)`. This and all predicates prefixed `kdbg_` are used to study the evolution of a process. When evaluated, the predicate displays a path of length `NumIter` in the extended LTS of process `KellProc`. At each step of the path both the action taken and resulting process after the action are displayed. Depending on `TxSemantics`, the path may include only τ transitions, when reduction semantics are specified, or it may include any type of transition otherwise.

- `kdbg_nth_acts(TxSemantics, KellProc, NumIter)`. Similar to the previous predicate but only actions along the path are displayed.

- `kdbg_nth_all(TxSemantics, KellProc, NumIter)`. All paths of length `NumIter` are displayed. Both actions taken in the transitions and resulting processes are displayed. If `TxSemantics` indicates reduction semantics, only τ transitions are considered. All three arguments are input arguments.

- `kdbg_all_acts(TxSemantics, KellProc, NumIter)`. Similar to the previous predicate, but only actions are displayed.

Prolog terms are used to represent kell-m processes. These terms are listed in Table 6.1. All the terms in the table with the exception of predicate `k_fresh` correspond to kell-m expressions. `k_fresh` is used to generate fresh unique names (cf. Section 3.3.1).

Term	kell-m Process
`zero`	$\mathbf{0}$
`k_nu(a, P)`	**new** $a\ P$
`k_fresh(a, P)`	$P\{n/a\}$, with n fresh
`k_par(P, Q)`	$P \mid Q$
`k_write(a, ws)`	$\overline{a}(\widetilde{ws})$
`k_wmatch(a, cs, P)`	$a(\widetilde{cs}) \triangleright P$
`k_rwmatch(a, cs, P)`	$a(\widetilde{cs}) \diamond P$
`k_kell(K, P)`	$K[P]$
`k_kmatch(K, X, P)`	$K[X] \triangleright P$
`k_rkmatch(K, X, P)`	$K[X] \diamond P$
`k_proc(P(ws))`	$P(\widetilde{ws})$
`k_def(P(cs), Q)`	$P(\widetilde{cs}) \stackrel{def}{=} Q$

Table 6.1: Prolog Terms for kell-m Processes

The tool uses Prolog's unification to implement the passing of values in communications. It requires restricted and fresh names to be specified as Prolog variables. All other names can be specified as atoms. Arguments `ws` and `cs` in the terms can be atoms, Prolog variables, and lists of atoms and variables. A τ transition occurs only if the name of the channel or kell in the abstraction matches the name of the concretion, and the parameters in the communication can be unified. Hence, concretion `k_write(a, [ foo, bar ])` can communicate with abstractions `k_wmatch(a, [X, Y], P)`, as well as `k_wmatch(a, [X, bar ])`, but not with `k_wmatch(a, [X], P)`.

For example, process $K[\overline{a}(w)] \mid \mathit{stop}(K) \triangleright K[X] \triangleright \mathbf{0}$ is represented in the kell-m checker as:

```
k_par(k_kell(K, k_write(a, [ w ])),
      k_wmatch(stop, [K], k_kmatch(K, X, zero)))
```

Table 6.2 lists the Prolog terms used in the tool to represent kμ formulas. Argument `S` in the terms corresponds to a list of action conditions `Tcnd`. Action conditions `Tcnd` correspond to the φ expressions specified in kμ formulas (cf. Section 4.2). The terms for the specification of action conditions are listed in Table 6.3. Kell containment conditions are represented as `Kcnd` in the terms. They correspond to the γ expressions in kμ formulas. For τ actions, `Kcnd_a` corresponds to γ_a, and `Kcnd_c` to γ_c.

The terms for `Kcnd` are listed in Table 6.4. `K` in the terms corresponds to a list of kell names and is represented as $\mathcal{K}$ in the kμ kell containment condition.

Recall the kμ formula used to identify if a time-to-live is still active (cf. Section 5.2.2):

$$(\mathit{Ttl} = \mathit{null}) \ \vee \ \mathbf{E}_e(\langle \overleftrightarrow{\mathit{Ttl}}(\mathit{Rc}) \rangle . \mathit{returns\_true}(\mathit{Rc}))$$

Term	kμ **Formula**
`tt`	`tt`
`ff`	`ff`
`kNot(F)`	$\neg\mathcal{F}$
`kPred(C, F)`	$\mathcal{C}.\mathcal{F}$
`kAnd(F1, F2)`	$\mathcal{F}1 \wedge \mathcal{F}2$
`kOr(F1, F2)`	$\mathcal{F}1 \vee \mathcal{F}2$
`kDiam(Tcnd, F)`	$\langle\varphi\rangle.\mathcal{F}$
`kDiamMinus(Tcnd, F)`	$\langle-\varphi\rangle.\mathcal{F}$
`kDiamSet(S, F)`	$\langle S\rangle.\mathcal{F}$
`kDiamSetMinus(S, F)`	$\langle-S\rangle.\mathcal{F}$
`kBox(Tcnd, F)`	$[\varphi].\mathcal{F}$
`kBoxMinus(Tcnd, F)`	$[-\varphi].\mathcal{F}$
`kBoxSet(S, F)`	$[\varphi].\mathcal{F}$
`kBoxSetMinus(S, F)`	$[S].\mathcal{F}$
`kPred(N(Ws))`	$N(\widetilde{Ws})$
`kDef(N(Cs),F)`	$N(\widetilde{Cs}) \stackrel{def}{=} \mathcal{F}$

Table 6.2: Prolog Terms for kμ Formulas

Such formula is represented in the kell-m checker as:

```
kOr(kPred((Ttl == null), tt),
    kEe(kDiam(kTCnd(tau(k_wmatch(Ttl, [Rc]), k_write(Ttl, [Rc])),
                    kcAny, kcAny),
              kForm(returns_true(Rc)))))
```

where `kEe` corresponds to $\mathbf{E}_e$.

6.3.1 Implementation Approach

The kell-m checker encodes kell-m extended LTS semantics using an extended version of the Mobility Model Checker MMC [3, 171] (also cf. Section 4.7.2). MMC is a tool for the verification of systems specified using a variation of the first-order synchronous π-calculus. MMC is written in XSB Prolog [11], a Prolog implementation with tabled evaluation [42]. MMC has allowed us to produce a kell-m model checker quickly. Nevertheless, kell-m system and property specifications are independent of MMC.

The kell-m checker translates kell-m process expressions into MMC process expressions. The translation is incremental and occurs as the process evolves. A callback feature in

Term	Action Condition
`k_write(a, ws), Kcnd`	$(\overline{a}(\widetilde{\mathrm{ws}}), \delta)$
`k_wmatch(a, cs), Kcnd`	$(a(\widetilde{\mathrm{cs}}), \delta)$
`k_kell(K, P), Kcnd`	$(\overline{K}(P), \delta)$
`k_kmatch(K, X), Kcnd`	$(K[X], \delta)$
`tau(k_wmatch(a, cs), Kcnd_a, k_write(a, ws), Kcnd_c)`	$(\overleftrightarrow{a}(\widetilde{ws}), \delta_a, \delta_c)$
`tau(k_kmatch(K, X), Kcnd_a, k_write(K, P), Kcnd_c)`	$(\overleftrightarrow{K}[\widetilde{P}], \delta_a, \delta_c)$

Table 6.3: Prolog Terms for Action Conditions

Term	Kell Containment Condition
`kcAny`	$*$
`kcNone`	$\emptyset$
`kcExactly(K)`	$\mathcal{K}$
`kcAtLeast(K)`	$\supseteq \mathcal{K}$
`kcExcept(K)`	$\nsubseteq \mathcal{K}$

Table 6.4: Prolog Terms for Kell Containment Conditions

MMC allows the invocation of code provided by the kell-m checker at specific points in the evolution of the process.

The main reasons for an incremental translation are the kell semantics and higher-order nature of kell-m. The traditional approach to deal with higher-order process expression has been to deduce behaviourally equivalent first-order expressions [144, 145]. We show, in Section 6.3.3, this approach is not compatible with kell-m semantics.

kμ formulas are translated to equivalent MMC property formulas. The model checking features in MMC are then used to verify the resulting MMC expressions and properties.

To deal with τ expressions exposing the participants in the communication and to be able to model check using reduction semantics we extended MMC. In the following sections we detail the translations from kell-m to MMC's π-calculus, and kμ to MMC's property language. We start with a brief description of MMC.

6.3.2 Mobility Model Checker

MMC represents π-calculus names as Prolog atoms and variables, and relies on Prolog's unification mechanism to pass parameter values to process definitions (e.g., $P(a, b, c)$), and

to pass values during communication (e.g., $\overline{a}(c).P \mid a(d).Q$). The use of Prolog unification for value passing allows the specification of expressions such as $\overline{a}(c,[d,e])$. In such expression, the values being output on channel a are name c and the list of names $[d,e]$. A matching abstraction is $a(x,\ y)$, with x and y variables. Upon communication x is unified to c, and y is unified to $[d,e]$. As we will show later, to encode the kell containment information required by kell-m's extended LTS semantics, we take advantage of MMC's ability to transmit lists in communications.

Conditions to be checked are represented in MMC using a subset of the $\pi\mu$-calculus [50]. A predicate `mck(P, C)` is used to check if a condition `C` is satisfied by process `P`. The actual calculus semantics are encoded by the predicate `trans/5`. `trans` can compute the set of transitions for π-calculus expressions not containing the replication operator !.

We call the process algebra in MMC the MMCπ calculus, and say a process expression is a MMCπ when it is a MMCπ calculus expression. The MMCπ calculus is based on the π-calculus, and is determined as follows:

$$P \quad ::= \quad \mathbf{0} \ \mid\ \mathbf{new}\ a\ P \ \mid\ P|P \ \mid\ P{+}P \ \mid\ \overline{a}(\widetilde{c}).P \ \mid\ a(\widetilde{c}).P \ \mid\ P(\widetilde{c}) \ \mid\ [a=b]P$$

With $\widetilde{c}$ representing zero or more names, and in general, any Prolog unifiable expression. We only use atoms, variables, and lists of atoms and variables. The *choice* operator $+$, represents a non-deterministic choice between two processes. $[a=b]P$ behaves as P if a and b are the same, otherwise it behaves as $\mathbf{0}$. $P(\widetilde{c})$ is process invocation, where P has been defined as $P(\widetilde{d}) \stackrel{def}{=} P_d$. Recursion is allowed in process definitions. Because recursion is allowed, there is no replication operation $!P$ (in the π-calculus, $P! \stackrel{def}{=} P \mid !P$, [116]).

A transition is denoted by $P \xrightarrow{M,\alpha} Q$, where P and Q are MMCπ expressions, α is τ, $\overline{a}(\widetilde{c})$, $\overline{r}(\widetilde{c})$, or $\overline{a}(\mathbf{new}\ \widetilde{c}\ )$. M is a set of equality constraints on names with syntax:

$$M \quad ::= \quad \emptyset \ \mid\ \{a=b\} \ \mid\ M \cup M$$

$\emptyset$ represents no conditions on names. $M_1 \cup M_2$ is sometimes written $M_1 M_2$.

The transition semantics implemented by the `trans` predicate are those specified by [98], and summarized in Figure 6.2. Right versions for the πSum, πPar, πClose, and πCom transitions are not shown, but are assumed. Functions *fn* and *bn* return the set of free and bound names in actions and process expressions. *fn* and *bn* were defined for kell-m in Section 3.4; a similar definition for MMCπ is assumed (details in [171]).

`trans(P, A, M, N, Q)`, represents a transition from `P` to `Q` where action `A` (α in the transition rules) has been taken. `M` is the set of constraints. `N` is a numeric parameter used internally by MMC to control whether unification of input and output channels is required to be stored in `M`, as specified for L in the πCom and πClose transitions. The predicate `trans` is used by the MMC checker to generate the LTS for a given process expression.

$$\pi\textsc{Pre}\ \frac{}{\alpha.P \xrightarrow{\emptyset,\alpha} P} \qquad \pi\textsc{Match}\ \frac{P \xrightarrow{M,\alpha} Q}{[a=b]P \xrightarrow{ML,\alpha} Q}\ L = \begin{cases} a = b \text{ if } a \neq b \\ \emptyset \qquad \text{otherwise} \end{cases}$$

$$\pi\textsc{Sum}\ \frac{P \xrightarrow{M,\alpha} Q}{P + R \xrightarrow{M,\alpha} Q} \qquad \pi\textsc{Par}\ \frac{P \xrightarrow{M,\alpha} Q}{P|R \xrightarrow{M,\alpha} Q|R}\ bn(\alpha) \cap fn(R) = \emptyset$$

$$\pi\textsc{Res}\ \frac{P \xrightarrow{M,\alpha} Q}{\textbf{new}\ c\ P \xrightarrow{M,\alpha} \textbf{new}\ c\ Q}\ c \notin names(M, \alpha)$$

$$\pi\textsc{Com}\ \frac{P \xrightarrow{M,a(c)} P', Q \xrightarrow{N,\overline{b}(d)} Q'}{P|Q \xrightarrow{MNL,\tau} P'\{d/x\}|Q'}\ L = \begin{cases} a = b \text{ if } a \neq b \\ \emptyset \qquad \text{otherwise} \end{cases}$$

$$\pi\textsc{Open}\ \frac{P \xrightarrow{M,\overline{a}(c)} Q}{\textbf{new}\ c\ P \xrightarrow{M,\overline{a}(\textbf{new}\ c\ )} Q}\ c \notin names(M, a)$$

$$\pi\textsc{Close}\ \frac{P \xrightarrow{M,a(c)} P', Q \xrightarrow{N,\overline{b}(\textbf{new}\ c\ )} Q'}{P|Q \xrightarrow{MNL,\tau} \textbf{new}\ c\ (P'|Q')}\ L = \begin{cases} a = b \text{ if } x \neq y \\ \emptyset \qquad \text{otherwise} \end{cases}$$

$$\pi\textsc{Ide}\ \frac{P_d\{\widetilde{c}/\widetilde{d}\} \xrightarrow{M,\alpha} Q}{P(\widetilde{c}) \xrightarrow{M,\alpha} Q}\ P(\widetilde{d}) \stackrel{def}{=} P_d$$

Figure 6.2: MMC Transition Semantics

Besides the π-calculus transitions specified above, a transition:

```
trans(code(Op, P), A, M, N, Q)
```

is included in MMC. Process expression `code(Op, P)` performs the Prolog predicate `Op`, and then behaves like `P`. This process expression, as we will see later, allows us to encode the kell-m calculus in MMC. The Prolog definition for this transition is:

```
trans(code(Op, P), A, M, N, Q) :- call(Op), trans(P, A, M, N, Q).
```

We extend MMC by exposing the channels and kells involved in τ transitions and by providing a `trans_tau`/5 predicate defined as:

```
trans_tau(P, tau(Rd, Wr)m V, N, Q) :- trans(P, tau(Rd, Wr), V, N, Q).
```

`tau(Rd, Wr)` corresponds to a τ transition where the actions are exposed. `Rd` is the abstraction involved in the communication and `Wr` is the concretion. For property verification using LTS semantics, predicate `trans` is used; for reduction semantics `trans_tau` is used instead.

6.3.3 Encoding kell-m in MMC

To encode the kell-m calculus in MMC two main issues need to be addressed: the semantics of kells, and the higher-order nature of the kell-m calculus.

As established by the semantics of kell-m (cf. Figure 3.4, and Section 3.4.1), at a given point, a kell may be non-deterministically passivated by a trigger, or it may continue its execution. When encoding kell-m in MMC, we use the name of the kell as a communication channel where the kell's own process is output. For example, a kell $K[P]$ is represented as:

$$choice(\overline{K}(P), K[advance(P)])$$

Where *choice* behaves as the π-calculus non-deterministic choice between two processes, and *advance* advances the execution of a process.

Higher-Order Expressions

MMC supports a variation of the first-order π-calculus. To deal with higher-order expressions in kell-m we could try Sangiorgi's approach, originally proposed to reduce higher-order π-calculus expressions to behaviour equivalent first-order π-calculus expressions [144, 145]. In Sangiorgi's approach, higher-order expressions are replaced by names we call *higher-order references*. Every higher-order expression is then composed in parallel and activated by a higher-order reference. The resulting first-order expression, could then be converted to a π-calculus expression.

Comparing with traditional programming languages, one can think of Sangiorgi's approach as the passing of function pointers instead of the function code itself. For example, consider the following higher-order kell-m process:

$$\overline{a}(\overline{b}(c)) \mid a(x) \triangleright (x \mid x)$$

The process is equivalent to a kell-m expression where $\overline{b}(c)$ is replaced with a fresh name h, and the higher-order expression is factorized into a process activated by channel h:

$$\textbf{new } h\ (\overline{a}(h) \mid h() \triangleright \overline{b}(c) \mid h() \triangleright \overline{b}(c)) \mid a(x) \triangleright (\overline{x}() \mid \overline{x}())$$

The fresh name h is, in essence, a reference to the factorized higher-order expression it replaced. Unfortunately, this approach is not compatible with kell semantics. Consider the following kell-m expression E_1, where P is a kell-m process:

$$E_1 \equiv \overline{a}(P) \mid a(x) \triangleright K[x]$$

Such an expressions is higher-order because process P is being output on channel a. Assuming Sangiorgi's approach applies to kell-m, such an expression would be equivalent to E_2:

$$E_2 \equiv \textbf{new } h\ (\overline{a}(h) \mid h() \triangleright P) | a(x) \triangleright K[\overline{\mathrm{x}}()]$$

After there is a match between $\overline{a}(h)$ and $a(x)$ in E_2, kell K causes the activation of P by executing $\overline{x}()$. P then executes *outside* kell K. In the original expression $a(x) \triangleright K[x]$ the intention is to execute the received kell-m expression P *inside* kell K. Because P executes inside K, another process can use a trigger $K[y] \triangleright Q$ to suspend K in the middle of P's execution. In E_2 there is no way to suspend the execution of P once it has been activated via h. The behaviour in E_2 is that of remote code invocation, whilst the behaviour in E_1 is similar to code migration and local execution of the received code. Clearly the semantics are different.

Another interesting feature related to the higher-order nature of the kell-m calculus is the scope extrusion of restricted names when kells are passivated. A process expression, being sent from one kell to another, may have restricted names that must remain restricted at the receiving kell. For example, in:

$$K_1[\textbf{new } c,d\ \overline{a}(\overline{c}(d) | c(e) \triangleright \mathbf{0})] \mid K_2[a(x) \triangleright x]$$

when $(\overline{c}(d) | c(e) \triangleright \mathbf{0})$ is received by K_2, names c and d should remain restricted. This scope extrusion is explicitly represented by the reduction rules *-ReductExtrusion presented in Section 3.4.2.

We call the *process context* of a higher-order process expression P, the set of restricted names for P at a given time. When there is code-shipping, the context of a given higher-order expression may vary depending on what code is actually received. For example, in:

$$\overline{a}(P) \mid \overline{a}(Q) \mid a(x) \triangleright a[x]$$

The context of the expression $a[x]$ depends on which one of P and Q is matched by $a(x)$. For some executions it may be P, for others it may be Q. Only until one analyses a

specific execution it is possible to determine which context the expression $a[x]$ took during the execution.

When code is not migrated but referenced, as it is the case in Sangiorgi's approach, there is no scope extrusion since the higher-order code is factorized into a process expression that shares the same context as the writing expression: $\overline{a}(P)$ is converted into **new** h $(\overline{a}(h) \mid h()\diamond P)$. When $\overline{h}()$, P executes within the same context as $\overline{a}(h)$.

Because of the issues with higher-order expression inside kells, and the need to carry-over processes along with their contexts when they are transmitted via channels, we present a *runable* interpretation of kell-m expressions as MMCπ expressions. We call it *runable*, because the resulting MMCπ expression is equivalent to a specific execution of the kell-m expression. We will show that the MMC encoding of a kell-m process simulates the kell-m process.

Our solution to deal with higher-order expressions in the MMC interpretation, is to replace the higher-order expressions with fresh names we call *higher-order indicators*. When a higher-order indicator is received by a trigger, the higher-order indicator is *replaced* by its associated kell-m expression. For example, in the following process:

$$\overline{a}(\overline{b}(c)) \mid a(x) \triangleright x$$

$\overline{b}(c)$ is replaced with a fresh name h:

$$\overline{a}(h) \mid \mathcal{H}[\![\ a(x)\ ]\!] \triangleright x \rightarrow \mathcal{H}[\![\ x\{h/x\}\ ]\!] \equiv \overline{b}(c)$$

Fresh name h is a higher-order indicator, and its relation to $\overline{b}(c)$ is remembered. When there is a communication, and the process reduces to $x\{h/x\} \equiv h$, h is then replaced by $\overline{b}(c)$. We call this process *instantiation of higher-order indicators*, and use $\mathcal{H}[\![\ \cdot\ ]\!]$ for its representation.

Fresh Names and Higher-Order Mappings

Before we define the MMCπ runable interpretation of kell-m processes, we introduce some auxiliary definitions. $\mathcal{F}[\![\ P\ ]\!]$ is defined in Figure 6.3. When Prolog variables are used care must be taken with expressions such as:

$$\textbf{new}\ c\ ((\textbf{new}\ c\ \overline{a}(c)) \mid \overline{b}(c))$$

Notice the c in $\overline{a}(c)$ is a different name than the c in $\overline{b}(c)$. If only one Prolog variable is used to represent c, when either $\overline{a}(c)$ or $\overline{b}(c)$ is instantiated, for example as part of a communication, the other c is instantiated as well. This problem is avoided by guaranteeing MMCπ expressions passed to MMC do not have name collisions.

$$
\begin{aligned}
\mathcal{F}[\![\ \mathbf{0}\ ]\!] &\stackrel{def}{=} \mathbf{0} \\
\mathcal{F}[\![\ x\ ]\!] &\stackrel{def}{=} x \\
\mathcal{F}[\![\ \mathbf{new}\ a\ P\ ]\!] &\stackrel{def}{=} \mathbf{new}\ v\ \mathcal{F}[\![\ P\{v/a\}\ ]\!] \text{ with } v \text{ a fresh name} \\
\mathcal{F}[\![\ \overline{a}(\widetilde{w})\ ]\!] &\stackrel{def}{=} \overline{a}(\mathcal{F}[\![\ \widetilde{w}\ ]\!]) \\
\mathcal{F}[\![\ \widetilde{w}\ ]\!] &\stackrel{def}{=} \mathcal{F}[\![\ w_1\ ]\!], ..., \mathcal{F}[\![\ w_n\ ]\!] \text{ where } \widetilde{w} \equiv w_1, ..., w_n \\
\mathcal{F}[\![\ K[P]\ ]\!] &\stackrel{def}{=} K[\mathcal{F}[\![\ P\ ]\!]] \\
\mathcal{F}[\![\ a(\widetilde{c}) \triangleright P\ ]\!] &\stackrel{def}{=} a(\widetilde{v}) \triangleright \mathcal{F}[\![\ P\{\widetilde{v}/\widetilde{c}\}\ ]\!] \text{ with } |\widetilde{v}| = |\widetilde{c}| \text{ and } v_i \in \widetilde{v} \text{ all fresh names} \\
\mathcal{F}[\![\ K[x] \triangleright P\ ]\!] &\stackrel{def}{=} K[v] \triangleright \mathcal{F}[\![\ P\{v/x\}\ ]\!] \text{ with } v \text{ a fresh name} \\
\mathcal{F}[\![\ P \mid Q\ ]\!] &\stackrel{def}{=} \mathcal{F}[\![\ P\ ]\!] \mid \mathcal{F}[\![\ Q\ ]\!] \\
\mathcal{F}[\![\ P(\widetilde{w})\ ]\!] &\stackrel{def}{=} P(\mathcal{F}[\![\ \widetilde{w}\ ]\!])
\end{aligned}
$$

Figure 6.3: Introduction of Fresh Names

Assuming H, the set of higher-order mappings in process expressions (initially empty), $\mathcal{H}[\![\ P\ ]\!]$ is defined as the instantiation of the higher-order indicators in a kell-m process P:

$$
\mathcal{H}[\![\ P\ ]\!] \quad \equiv \quad P\{\widetilde{w}/\widetilde{h}\} \text{ with } |\widetilde{w}| = |\widetilde{h}|, \text{ and } \begin{cases} w_i = p_i, & \text{if } \exists p_i : (h_i, p_i) \in H \\ w_i = h_i, & \text{otherwise} \end{cases}
$$

$\mathcal{H}$ is defined recursively in Figure 6.4.

$$
\begin{aligned}
\mathcal{H}[\![\ \mathbf{0}\ ]\!] &\stackrel{def}{=} \mathbf{0} \\
\mathcal{H}[\![\ h\ ]\!] &\stackrel{def}{=} \begin{cases} Q & \text{if } (h, Q) \in H \\ h & \text{otherwise} \end{cases} \\
\mathcal{H}[\![\ \mathbf{new}\ a\ P\ ]\!] &\stackrel{def}{=} \mathbf{new}\ a\ \mathcal{H}[\![\ P\ ]\!] \\
\mathcal{H}[\![\ a(\widetilde{b}) \triangleright P\ ]\!] &\stackrel{def}{=} a(\widetilde{b}) \triangleright \mathcal{H}[\![\ P\ ]\!] \\
\mathcal{H}[\![\ K[x] \triangleright P\ ]\!] &\stackrel{def}{=} K[x] \triangleright \mathcal{H}[\![\ P\ ]\!] \\
\mathcal{H}[\![\ K[P]\ ]\!] &\stackrel{def}{=} K[\mathcal{H}[\![\ P\ ]\!]] \\
\mathcal{H}[\![\ \overline{a}(w_1, ..., w_n)\ ]\!] &\stackrel{def}{=} \overline{a}(w_1, ..., w_n) \\
\mathcal{H}[\![\ P(\widetilde{w})\ ]\!] &\stackrel{def}{=} \mathcal{H}[\![\ P_d\{\widetilde{w}/\widetilde{y}\}\ ]\!] \text{ with } P(\widetilde{y}) \stackrel{def}{=} P_d \\
\mathcal{H}[\![\ P \mid Q\ ]\!] &\stackrel{def}{=} \mathcal{H}[\![\ P\ ]\!] \mid \mathcal{H}[\![\ Q\ ]\!]
\end{aligned}
$$

Figure 6.4: Instantiation of Higher-Order Indicators

Kell Passivation

To implement kell-m extended semantics, it is necessary to keep track of kell containment information. A set K_s of kells is used for this purpose. When a kell-m abstraction or

$$
\begin{aligned}
\mathcal{S}[\![\ R, B, K[P], K_s\ ]\!] &\stackrel{def}{=} \mathit{Hwrite}(\overline{K}(P, K_s), \mathcal{I}[\![\ B, R\{\mathbf{0}/K[P]\}\ ]\!], B \setminus \{K\}) \\
&\quad + \begin{cases} \mathcal{S}[\![\ R, B, P, K_s \cup \{K\}\ ]\!] & \text{if } R \neq \mathbf{0} \\ \mathcal{S}[\![\ K[P], B, P, K_s \cup \{K\}\ ]\!] & \text{otherwise} \end{cases} \\
\mathcal{S}[\![\ R, B, P \mid Q, K_s\ ]\!] &\stackrel{def}{=} \mathcal{S}[\![\ R, B, P, K_s\ ]\!] \ +\ \mathcal{S}[\![\ R, B, Q, K_s\ ]\!] \\
\mathcal{S}[\![\ R, B, P, K_s\ ]\!] &\stackrel{def}{=} \mathbf{0} \text{ if } P \neq Q \mid R, \text{ and } P \neq K[Q] \\
\mathit{Hwrite}(\overline{a}(\widetilde{w}, K_s), Q, B) &\stackrel{def}{=} \mathit{Hconv}(\overline{a}(\widetilde{w}, K_a, K_s, B).Q, |\widetilde{w}|, 1), \text{ with } K_a \text{ a variable}
\end{aligned}
$$

$$
\mathit{Hconv}(P, n, i) \stackrel{def}{=} \begin{cases} P & \text{if } i \geq n \\ \mathit{Hconv}(\mathbf{new}\ h\ (P\{h/w_i\}), n, i+1), \text{ with } h \text{ fresh and} & \\ \qquad H := H \cup \{(h, w_i)\} & \\ \qquad \text{if } w_i \text{ is not a name} & \\ \mathit{Hconv}(P, n, i+1) & \text{otherwise} \end{cases}
$$

Figure 6.5: Encoding of Kells

concretion is to be encoded in MMC, the input or output parameters are extended with the kell containment information.

A kell $K[P]$ can be passivated by a matching trigger $K[x] \triangleright Q$, or it can advance its execution. The choice between passivating the kell and advancing the execution of its process is non-deterministic. A kell $K[P]$ itself may be nested within another kell-m process. For example, consider $R = K_1[K_2[K[P]]]$.

We specify $\mathcal{S}[\![\ R, B, K[P], K_s\ ]\!]$ as the MMCπ process corresponding to the passivation of a kell $K[P]$ nested in process R, executing within kells K_s, and with restricted names B (cf. Figure 6.5).

Hwrite replaces higher-order expressions in a channel output with higher-order indicators. For example, $\mathit{Hwrite}(\overline{a}(\overline{b}(c).\mathbf{0}).\mathbf{0})$ produces the expression $\mathbf{new}\ h\ (\overline{a}(h).\mathbf{0})$, where $H := H \cup \{(h, \overline{b}(c))\}$.

$\mathcal{I}[\![\ B, P\ ]\!]$, to be defined later, is the MMC π-calculus interpretation of P when B represents the set of restricted names at the time P is interpreted.

Since higher-order process expressions can be output on channels, we need to include the restricted names in the process expressions as well. As previously mentioned, we refer to these restricted names, as the *context* of the process expressions output. Here we are abusing the notation by writing the set of kells K_s and restricted names B directly. Alternatively, one can assume an arbitrarily large number of names being passed in the concretions and being received in the abstractions. The first names would correspond to the elements of K_s and B, the rest of the names would be special names used to specify when a position is not being used. For notational convenience, we specify the set of kells

and bound names directly. We extend this abuse of notation to kell-m channel abstractions and concretions. Therefore, we write $\overline{a}(\widetilde{w}, K_s)$ and $a(\widetilde{c}, K_s)$, with K_s kell containment sets.

Kell-m concretions $\overline{a}(\widetilde{w})$ are encoded as MMCπ expressions $\overline{a}(\widetilde{w}, K_a, K_c, B)$. As we will show later, abstractions $a(\widetilde{c})$ are encoded as $a(\widetilde{c}, K_a, K_c, B)$. If a concretion, K_c is the kell containment set for the action and K_a is a Prolog variable which is instantiated at the time of a τ transition involving the channel a. If an abstraction, K_a is the kell containment set, and K_c and B are variables instantiated at τ transitions. Hence, we are taking advantage of Prolog's unification for exposing kell containment information of both, concretions and abstractions.

Consider the kell-m process:

$$K_1[K_2[K_3[a(\widetilde{c}) \triangleright \mathbf{0}] \mid \overline{a}(\widetilde{w})]]$$

$\mathcal{S}$ generates MMCπ code allowing the passivation of any of the three kells. Let $Q = K_2[K_3[a(\widetilde{c}) \triangleright \mathbf{0}] \mid \overline{a}(\widetilde{w})]$, $R = K_3[a(\widetilde{c}) \triangleright \mathbf{0}] \mid \overline{a}(\widetilde{w})$, and $P = K_1[Q]$. By $\mathcal{S}$'s definition, the process is encoded as shown in Figure 6.6. In the resulting process expression, the passivated kell, if any, is replaced by the null process in P. If τ transitions, K_a, K'_a, and K''_a in Figure 6.6, are variables to be instantiated with kell containment sets of matching kell passivation triggers.

Advancing Kell Processes

If a kell $K[P]$ is not passivated, its process advances its execution as defined by $\mathcal{A}[\![\ K[P]\ ]\!]$ in Figure 6.7(a). As mentioned at the beginning of Section 6.3.3 kell abstractions are converted to channel abstractions. For concretions and abstractions on channels ($\overline{a}(\widetilde{w})$, $a(\widetilde{c})$), and abstractions on kells ($K[x]$), a kell containment set, initially empty, is added to the parameters of the communication. For kell concretions ($K[P]$), the kell containment set is added when the code allowing the kell passivation is generated by $\mathcal{S}[\![\ .\ ]\!]$.

Notice the re-invocation of $\mathcal{A}$ in the definition for $\mathcal{A}[\![\ K[L[P]]\ ]\!]$. The other, alternate but incorrect, definition is:

$$\mathcal{A}[\![\ K[L[P]]\ ]\!] \stackrel{def}{=} K[\mathcal{A}[\![\ L[P]\ ]\!]]$$

In the case of nested kells, this definition may cause the interruption of one of the kells before the execution has advanced. In other words, we want to lift the abstraction and concretion operations from nested kells in one single step. For example, consider the process:

$$K_1[K_2[K_3[\overline{a}(\widetilde{w})]]]$$

$$
\begin{aligned}
\mathcal{S}[\![\ \mathbf{0}, B, P, K_s\ ]\!] \equiv\ & \mathit{Hwrite}(\overline{K_1}(Q, K_s), \mathcal{I}[\![\ B, \mathbf{0}\ ]\!], B \setminus \{K_1\})\ +\ \mathcal{S}[\![\ P, B, Q, K_s \cup \{K_1\}\ ]\!] \\
\equiv\ & \mathbf{new}\ h_1\ \overline{K_1}(h_1, K_s, K_a, B \setminus \{K_1\}).\mathcal{I}[\![\ B, \mathbf{0}\ ]\!] \\
& +\ \mathit{Hwrite}(\overline{K_2}(R, K_s \cup \{K1\}), \mathcal{I}[\![\ B, K_1[\mathbf{0}]\ ]\!], B \setminus \{K_2\}) \\
& +\ \mathcal{S}[\![\ P, B, R, K_s \cup \{K_1, K_2\}\ ]\!], \text{ with } H := H \cup \{(h_1, Q)\} \\
\equiv\ & \mathbf{new}\ h_1\ \overline{K_1}(h_1, K_s, K_a, B \setminus \{K_1\}).\mathcal{I}[\![\ B, \mathbf{0}\ ]\!] \\
& +\ \mathbf{new}\ h_2\ \overline{K_2}(h_2, K_s \cup \{K_1\}, K'_a, B \setminus \{K_2\}).\mathcal{I}[\![\ B, K_1[\mathbf{0}]\ ]\!] \\
& +\ \mathcal{S}[\![\ P, B, K_3[a(\widetilde{c}) \triangleright \mathbf{0}], K_s \cup \{K_1, K_2\}\ ]\!] \\
& +\ \mathcal{S}[\![\ P, B, \overline{a}(\widetilde{w}), K_s \cup \{K_1, K_2\}\ ]\!], \\
& \text{with } H := H \cup \{(h_2, R)\} \\
\equiv\ & \mathbf{new}\ h_1\ \overline{K_1}(h_1, K_s, K_a, B \setminus \{K_1\}).\mathcal{I}[\![\ B, \mathbf{0}\ ]\!] \\
& +\ \mathbf{new}\ h_2\ \overline{K_2}(h_2, K_s \cup \{K_1\}, K'_a, B \setminus \{K_2\}).\mathcal{I}[\![\ B, K_1[\mathbf{0}]\ ]\!] \\
& +\ \mathit{Hwrite}(\overline{K_3}(a(\widetilde{c}, K_s \cup \{K_1, K_2\}) \triangleright \mathbf{0}), \mathcal{I}[\![\ B, K_1[K_2[\mathbf{0} \mid \overline{a}(\widetilde{w})]]\ ]\!], \\
& \qquad\qquad B \setminus \{K_3\}) \\
& +\ \mathcal{S}[\![\ P, B, a(\widetilde{c}) \triangleright \mathbf{0}, K_s \cup \{K_1, K_2, K_3\}\ ]\!]\ +\ \mathbf{0} \\
\equiv\ & \mathbf{new}\ h_1\ \overline{K_1}(h_1, K_s, K_a, B \setminus \{K_1\}).\mathcal{I}[\![\ B, \mathbf{0}\ ]\!] \\
& +\ \mathbf{new}\ h_2\ \overline{K_2}(h_2, K_s \cup \{K_1\}, K'_a, B \setminus \{K_2\}).\mathcal{I}[\![\ B, K_1[\mathbf{0}]\ ]\!] \\
& +\ \mathbf{new}\ h_3\ \overline{K_3}(h_3, K_s \cup \{K_1, K_2\}, K''_a, B \setminus \{K_3\}). \\
& \qquad\qquad \mathcal{I}[\![\ B, K_1[K_2[\mathbf{0} \mid \overline{a}(\widetilde{w})]]\ ]\!] \\
& +\ \mathcal{S}[\![\ P, B, a(\widetilde{c}) \triangleright \mathbf{0}, K_s \cup \{K_1, K_2, K_3\}\ ]\!]\ +\ \mathbf{0}, \\
& \text{with } H := H \cup \{(h_3, a(\widetilde{c}) \triangleright \mathbf{0})\} \\
\equiv\ & \mathbf{new}\ h_1\ \overline{K_1}(h_1, K_s, K_a, B \setminus \{K_1\}).\mathcal{I}[\![\ B, \mathbf{0}\ ]\!] \\
& +\ \mathbf{new}\ h_2\ \overline{K_2}(h_2, K_s \cup \{K_1\}, K'_a, B \setminus \{K_2\}).\mathcal{I}[\![\ B, K_1[\mathbf{0}]\ ]\!] \\
& +\ \mathbf{new}\ h_3\ \overline{K_3}(h_3, K_s \cup \{K_1, K_2\}, K''_a, B \setminus \{K_3\}). \\
& \qquad\qquad \mathcal{I}[\![\ B, K_1[K_2[\mathbf{0} \mid \overline{a}(\widetilde{w})]]\ ]\!] \\
& +\ \mathbf{0}\ +\ \mathbf{0}
\end{aligned}
$$

Figure 6.6: Sample Encoding of Nested Kells

By $\mathcal{A}$'s definition,

$$\mathcal{A}[\![\ K_1[K_2[K_3[\overline{a}(\widetilde{w})]]]\ ]\!] \equiv \overline{a}(\widetilde{w}, \{K_1, K_2, K_3\})$$

If the alternate incorrect definition for $\mathcal{A}$ is used, we obtain:

$$\mathcal{A}[\![\ K_1[K_2[K_3[\overline{a}(\widetilde{w})]]]\ ]\!] \equiv K_1[K_2[\overline{a}(\widetilde{w}, \{K_3\})]]$$

As we will see later, when we define $\mathcal{I}$, this alternate definition would allow the passivation of kells K_1 and K_2 at this point, at which time the kell processes do not correspond to the original ones ($K_3[\overline{a}(\widetilde{w})]$ in the case of K_2, and $K_2[K_3[\overline{a}(\widetilde{w})]]$ in the case of K_1).

$$
\begin{aligned}
\mathcal{A}[\![\ K[\mathbf{0}]\ ]\!] &\stackrel{def}{=} \mathbf{0} \\
\mathcal{A}[\![\ K[\mathbf{new}\ c\ P]\ ]\!] &\stackrel{def}{=} \mathbf{new}\ c\ \mathcal{A}[\![\ K[P]\ ]\!] \\
\mathcal{A}[\![\ K[\overline{a}(\widetilde{w})]\ ]\!] &\stackrel{def}{=} \mathcal{A}[\![\ K[\overline{a}(\widetilde{w}, \emptyset)]\ ]\!] \\
\mathcal{A}[\![\ K[\overline{a}(\widetilde{w}, K_s)]\ ]\!] &\stackrel{def}{=} \overline{a}(\widetilde{w}, K_s \cup \{K\}) \\
\mathcal{A}[\![\ K[a(\widetilde{c}) \triangleright P]\ ]\!] &\stackrel{def}{=} \mathcal{A}[\![\ K[a(\widetilde{c}, \emptyset) \triangleright P]\ ]\!] \\
\mathcal{A}[\![\ K[a(\widetilde{c}, K_s) \triangleright P]\ ]\!] &\stackrel{def}{=} a(\widetilde{c}, K_s \cup \{K\}) \triangleright K[P] \\
\mathcal{A}[\![\ K[L[x] \triangleright P]\ ]\!] &\stackrel{def}{=} L(x, \emptyset) \triangleright K[P] \\
\mathcal{A}[\![\ K[L[P]]\ ]\!] &\stackrel{def}{=} \mathcal{A}[\![\ K[\mathcal{A}[\![\ L[P]\ ]\!]]\ ]\!] \\
\mathcal{A}[\![\ K[P(\widetilde{w})]\ ]\!] &\stackrel{def}{=} \mathcal{A}[\![\ K[P_d\{\widetilde{w}/\widetilde{y}\}]\ ]\!] \text{ where } P(\widetilde{y}) \stackrel{def}{=} P_d
\end{aligned}
$$

(a) Simple Process

$$
\begin{aligned}
\mathcal{A}[\![\ K[\mathbf{0} \mid Q]\ ]\!] &\stackrel{def}{=} \mathcal{A}[\![\ K[Q]\ ]\!] \\
\mathcal{A}[\![\ K[\mathbf{new}\ c\ P \mid Q]\ ]\!] &\stackrel{def}{=} \mathbf{new}\ c\ \mathcal{A}[\![\ K[P \mid Q]\ ]\!] \\
\mathcal{A}[\![\ K[\overline{a}(\widetilde{w}) \mid Q]\ ]\!] &\stackrel{def}{=} \mathcal{A}[\![\ K[\overline{a}(\widetilde{w}, \emptyset) \mid Q]\ ]\!] \\
\mathcal{A}[\![\ K[\overline{a}(\widetilde{w}, K_s) \mid Q]\ ]\!] &\stackrel{def}{=} \overline{a}(\widetilde{w}, K_s \cup \{K\}) \mid K[Q] \\
\mathcal{A}[\![\ K[(a(\widetilde{c}) \triangleright P) \mid Q]\ ]\!] &\stackrel{def}{=} \mathcal{A}[\![\ K[(a(\widetilde{c}, \emptyset) \triangleright P) \mid Q]\ ]\!] \\
\mathcal{A}[\![\ K[(a(\widetilde{c}, K_s) \triangleright P) \mid Q]\ ]\!] &\stackrel{def}{=} a(\widetilde{c}, K_s \cup \{K\}) \triangleright K[P \mid Q] \\
\mathcal{A}[\![\ K[(L[x] \triangleright P) \mid Q]\ ]\!] &\stackrel{def}{=} \mathcal{A}[\![\ K[(L(x, \emptyset) \triangleright P) \mid Q]\ ]\!] \\
\mathcal{A}[\![\ K[L[P] \mid Q]\ ]\!] &\stackrel{def}{=} \mathcal{A}[\![\ K[\mathcal{A}[\![\ L[P]\ ]\!] \mid Q]\ ]\!] \\
\mathcal{A}[\![\ K[P(\widetilde{w}) \mid Q]\ ]\!] &\stackrel{def}{=} \mathcal{A}[\![\ K[P_d\{\widetilde{w}/\widetilde{y}\} \mid Q]\ ]\!] \text{ where } P(\widetilde{y}) \stackrel{def}{=} P_d
\end{aligned}
$$

(b) Composed Process

Figure 6.7: Advancing the Execution of a Non-passivated Kell

Assuming no name collisions, $\mathcal{A}[\![\ K[P|Q]\ ]\!]$ is defined in Figure 6.7(b). The assumption of no name collisions is important to avoid unintended capturing of names when lifting the name restrictions outside the kells ($\mathcal{A}[\![\ K[\mathbf{new}\ c\ P \mid Q]\ ]\!]$).

Notice there is no definition of $\mathcal{A}[\![\ x\ ]\!]$, with x a process variable. This is because the execution of a kell process can only advance when process variables have been replaced by their corresponding process expressions.

MMCπ Calculus Interpretation of kell-m Processes

The runable MMC π-calculus interpretation of a kell-m process P_k is defined as:

$$\mathit{Interpret}(P_k) \stackrel{def}{=} \mathcal{I}[\![\ \{\}, \mathcal{F}[\![\ P_k\ ]\!]\ ]\!], \text{where } \mathit{fn}(P_k) = \emptyset$$

fn is the set of free names in a process. $\mathcal{F}[\![\ P\ ]\!]$ is process P with fresh names. $\mathcal{I}[\![\ B, P\ ]\!]$ is the MMC π-calculus interpretation of P when B represents the set of restricted names at the time P is interpreted.

$\mathcal{I}[\![\ B,\ P\ ]\!]$ is defined in Figure 6.8. By passing the context along with the process expressions (see the definition for $\mathcal{I}[\![\ B, \overline{a}(\widetilde{w})\ ]\!]$ above) we guarantee any restricted names in the process expressions remain restricted at the receiving end of the communication. To avoid unintended capture of free names on the receiving process, $\mathcal{F}$ is used every time a process is invoked and every time there is communication.

$$\begin{aligned}
\mathcal{I}[\![\ B, \mathbf{0}\ ]\!] &\stackrel{def}{=} \mathbf{0} \\
\mathcal{I}[\![\ B, x\ ]\!] &\stackrel{def}{=} \mathbf{0} \\
\mathcal{I}[\![\ B, \mathbf{new}\ a\ P\ ]\!] &\stackrel{def}{=} \mathbf{new}\ a\ \mathcal{I}[\![\ B \cup \{a\}, P\ ]\!] \\
\mathcal{I}[\![\ B, \overline{a}(\widetilde{w})\ ]\!] &\stackrel{def}{=} \mathcal{I}[\![\ B, \overline{a}(\widetilde{w}, \emptyset)\ ]\!] \\
\mathcal{I}[\![\ B, \overline{a}(\widetilde{w}, K_s)\ ]\!] &\stackrel{def}{=} \mathit{Hwrite}(\overline{a}(\widetilde{w}, K_s), \mathbf{0}, B \setminus \{a\}) \\
\mathcal{I}[\![\ B, a(\widetilde{c}) \triangleright P\ ]\!] &\stackrel{def}{=} \mathcal{I}[\![\ B, a(\widetilde{c}, \emptyset) \triangleright P\ ]\!] \\
\mathcal{I}[\![\ B, a(\widetilde{c}, K_s) \triangleright P\ ]\!] &\stackrel{def}{=} a(\widetilde{c}, K_s, K_c, \mathit{bnd}).\texttt{code}(\texttt{inst}(P, \widetilde{c}, \mathit{bnd}, P_\pi), P_\pi) \\
&\quad \text{with } \texttt{inst}(P, \widetilde{c}, \mathit{bnd}, P_\pi)\ \texttt{:-}\ P_\pi = \mathcal{I}[\![\ B \cup \mathit{bnd}, \mathcal{F}[\![\ \mathcal{H}[\![\ P\ ]\!]\ ]\!]\ ]\!] \\
&\quad \text{and } K_c \text{ a variable} \\
\mathcal{I}[\![\ B, K[P]\ ]\!] &\stackrel{def}{=} \mathcal{S}[\![\ \mathbf{0}, B, K[P], \emptyset\ ]\!] \\
&\quad + \begin{cases} \mathcal{I}[\![\ B, \mathcal{A}[\![\ K[Q|R]\ ]\!]\ ]\!] + \mathcal{I}[\![\ B, \mathcal{A}[\![\ K[R|Q]\ ]\!]\ ]\!] & \text{if } P \equiv Q|R \\ \mathcal{I}[\![\ B, \mathcal{A}[\![\ K[P]\ ]\!]\ ]\!] & \text{otherwise} \end{cases} \\
\mathcal{I}[\![\ B, K[x] \triangleright P\ ]\!] &\stackrel{def}{=} \mathcal{I}[\![\ B, K(x, \emptyset) \triangleright P\ ]\!] \\
\mathcal{I}[\![\ B, P(\widetilde{w})\ ]\!] &\stackrel{def}{=} \mathcal{I}[\![\ B, \mathcal{F}[\![\ P_d\{\widetilde{w}/\widetilde{y}\}\ ]\!]\ ]\!] \text{ where } P(\widetilde{y}) \stackrel{def}{=} P_d \\
\mathcal{I}[\![\ B, P \mid Q\ ]\!] &\stackrel{def}{=} \mathcal{I}[\![\ B, P\ ]\!] \mid \mathcal{I}[\![\ B, Q\ ]\!]
\end{aligned}$$

Figure 6.8: Encoding of kell-m Processes into MMC's π-calculus

Similarly to the definition of $\mathcal{A}$, kell containment sets are added for concretions and abstractions on channels ($\overline{a}(\widetilde{w})$, $a(\widetilde{c})$), and for abstractions on kells ($K[x]$). In our tool $\emptyset$ is represented as an empty Prolog list [ ].

Recall *Hwrite* replaces higher-order expressions in a channel output with higher-order indicators. The resulting MMCπ expression corresponds to the channel output with higher-order expressions replaced by higher-order indicator names, plus the process scope, followed by a given MMCπ ($\mathbf{0}$ in the definition for $\mathcal{I}[\![\ B, \overline{a}(\widetilde{w})\ ]\!]$).

It is in the definition for $\mathcal{I}[\![\ B, a(\widetilde{c}) \triangleright P\ ]\!]$, that we make use of the MMC π-calculus extension `code(Op, P)`. As previously mentioned, process expression `code(Op, P)` performs the Prolog predicate `Op`, and then behaves like `P`. In the case of the kell-m encoding, `code(inst(`P`,` $\widetilde{c}$`,` bnd`,` P_π`),`P_π`)`, the arguments received in the channel (if any), are used in the replacement of higher-order indicators.

When the process being encoded is the parallel composition of two processes within a kell ($\mathcal{I}[\![\ B, K[Q|R]\ ]\!]$), we assume, non-deterministically, either one can advance. Hence we are using extensional (interleaved) concurrency: we model concurrent behaviour by, non-deterministically, interleaving the actions of parallel processes [114].

Notice there is a definition for $\mathcal{I}[\![\ B,\ x\ ]\!]$, with x a process variable. A MMCπ expression cannot be produced for a process variable until the variable has been instantiated with a higher-order indicator, and the higher-order indicator has been replaced by its associated process expression. Higher-order indicators are determined when the writer and receiver processes communicate on a channel, via a τ transition. If higher-order indicators associated to process variables are not known, the result of the interpretation is the null process. For example, the kell-m expression $a(x) \triangleright x$ is encoded as $a(x).\mathbf{0}$.

In Appendix B, we show for a kell-m process P: (a) the result of $\mathcal{I}[\![\ \{\},\ P\ ]\!]$, is a MMC π-calculus process (cf. B.1); and (b) a global observer cannot distinguish P from $\mathcal{I}[\![\ \{\},\ P\ ]\!]$ (cf. B.2).

6.3.4 Encoding kμ in MMC

Recall the language used in MMC for property specification. It was introduced in Section 4.7.2 and it is depicted again in Figure 6.9. α represents transition actions, S is a set of π-calculus actions, $\mathcal{V}$ is a set of names, $\mathcal{N}$ is a set of formula names, and $\mathcal{Z}$ is a set of formula variables.

`neg_form` is used to negate a condition specified by a formula. For a process `P`, formula `F`, and equality condition `Cond` on names, goal `mck(P, pred(Cond, F))` succeeds if `Cond` holds and `mck(P, F)`. Diamond, box and set modalities have the usual meaning. `fDef`, is used to name formulas and, along with `form`, allow recursive definitions.

For notational convenience we write $\mathcal{F} \wedge \mathcal{F}$ and $\mathcal{F} \vee \mathcal{F}$ instead of `fAnd(`$\mathcal{F}, \mathcal{F}$`)` and `fOr(`$\mathcal{F}, \mathcal{F}$`)`. Similarly we write $\neg\mathcal{N}(\mathcal{V})$ for `neg_form(`$\mathcal{N}(\mathcal{V})$`)`; $\mathcal{N}(\widetilde{\mathcal{Z}}) \stackrel{def}{=} \mathcal{F}$ for `fDef(`$\mathcal{N}(\widetilde{\mathcal{Z}}), \mathcal{F}$`)`; $\langle\alpha\rangle.\mathcal{F}$ for `fDiam(`$\alpha, \mathcal{F}$`)`; and $[\alpha].\mathcal{F}$ for `fBox(`$\alpha, \mathcal{F}$`)`. Idem for set diamond and set box modalities.

$$\begin{aligned}\mathcal{F} ::= \; & \texttt{tt} \mid \texttt{ff} \mid \texttt{neg\_form}(\mathcal{N}(\widetilde{\mathcal{V}})) \mid \texttt{pred(Cond}, \mathcal{F}) \mid \texttt{fAnd}(\mathcal{F}, \mathcal{F}) \mid \texttt{fOr}(\mathcal{F}, \mathcal{F}) \mid \\ & \texttt{fDiam}(\alpha, \mathcal{F}) \mid \texttt{fDiamMinus}(\alpha, \mathcal{F}) \mid \texttt{fDiamSet}(S, \mathcal{F}) \mid \\ & \texttt{fDiamSetMinus}(S, \mathcal{F}) \mid \texttt{fBox}(\alpha, \mathcal{F}) \mid \texttt{fBoxMinus}(\alpha, \mathcal{F}) \mid \texttt{fBoxSet}(S, \mathcal{F}) \mid \\ & \texttt{fBoxSetMinus}(S, \mathcal{F}) \mid \texttt{fDef}(\mathcal{N}(\widetilde{\mathcal{Z}}), \mathcal{F}) \mid \texttt{form}(\mathcal{N}(\widetilde{\mathcal{V}}))\end{aligned}$$

Figure 6.9: Syntax of Property Formulas in the Mobility Model Checker

$$\begin{aligned}\mathcal{E}_p(I, \kappa) &\stackrel{def}{=} \texttt{pred}(I := \kappa, \texttt{tt}) \\ \mathcal{E}_p(*, \kappa) &\stackrel{def}{=} \texttt{tt} \\ \mathcal{E}_p(\mathcal{K}, \kappa) &\stackrel{def}{=} \texttt{pred}(\mathcal{K} = \kappa, \texttt{tt}) \\ \mathcal{E}_p(\supseteq \mathcal{K}, \kappa) &\stackrel{def}{=} \texttt{pred}(\mathcal{K} \setminus \kappa = \emptyset, \texttt{tt}) \\ \mathcal{E}_p(\not\subseteq \mathcal{K}, \kappa) &\stackrel{def}{=} \texttt{pred}(\mathcal{K} \cap \kappa = \emptyset, \texttt{tt})\end{aligned}$$

Figure 6.10: Encoding of Kell Containment Conditions

Kell Containment Conditions

A kell containment condition γ in kμ has one of the following forms, where I is a variable and $\mathcal{K}$ is a set of kells (cf. Section 4.2) :

$$\gamma \quad ::= \quad * \mid \mathcal{K} \mid \supseteq \mathcal{K} \mid \not\subseteq \mathcal{K} \mid I$$

We start by defining, in Figure 6.10, function $\mathcal{E}_p$ for the translation of kell containment conditions. To avoid confusion with the functions defined in Section 6.3.3, the functions defined for the encoding of kμ have a suffix $_p$.

The function has as arguments a kμ kell containment condition γ and a list of kells K_s. K_s is the actual set of kells where a communication action is executing. The return value is a MMC formula of the form `tt` or `pred(Cond, tt)`.

When a variable I is specified it is instantiated with the set of kells K_s. The other possible values are directly deduced from the kμ semantics (cf. Section 4.4 and Figure 4.6).

For readability we use set operators in the definition. The implementation of $\mathcal{E}_p$ in the kell-m checker has the set operations, shown as arguments to predicate `pred`, replaced with the corresponding Prolog set predicates.

$$
\begin{aligned}
\mathcal{A}_p(\overline{a}(\widetilde{w})) &\stackrel{def}{=} \overline{a}(\widetilde{w}, K_a, K_c, B) \\
\mathcal{A}_p(a(\widetilde{c})) &\stackrel{def}{=} a(\widetilde{c}, K_a, K_c, B) \\
\mathcal{A}_p(\overline{k}[h]) &\stackrel{def}{=} \overline{k}(h, K_a, K_c, B) \\
\mathcal{A}_p(k[x]) &\stackrel{def}{=} k(x, K_a, K_c, B) \\
\mathcal{A}_p(\overleftrightarrow{a}(\widetilde{w})) &\stackrel{def}{=} \tau(\overline{a}(\widetilde{w}, K_a, K_c, B)) \\
\mathcal{A}_p(\overleftrightarrow{k}[h]) &\stackrel{def}{=} \tau(\overline{k}(h, K_a, K_c, B))
\end{aligned}
$$

Figure 6.11: Encoding of Actions

Actions

In Figure 6.11 we define $\mathcal{A}_p$, a function for the translation of kell-m actions into MMCπ actions. The function receives as its only argument an action, α or α_τ, and returns the corresponding MMC action. Recall α represents abstraction and concretion transitions; α_τ represents τ transitions (e.g., $\overleftrightarrow{a}(\widetilde{w})$).

Because kells are encoded in MMC using regular channels (cf. Section 6.3.3), kell abstractions and concretions in kell-m correspond to channel concretions in MMC. τ transitions are encoded as MMC τ transitions extended to expose the channel or kell involved in the communication and the parameters of the communication. In all MMC actions, kell containment sets K_a and K_c are included in the parameters of the communication as well as the set of bound names B. Sets are implemented as lists in the encoding.

For abstractions, K_c and B correspond to Prolog variables, and K_a to the set of kells where the abstraction is executing. For concretions K_a is a Prolog variable, K_c is the set of kells where the concretion is located, and B contains the bound names of the process.

When τ transitions, all K_a, K_c and B are sets. K_a is the kell containment set of the matching abstraction; K_c is the kell containment set of the matching concretion; and B is the set of bound names of the concretion.

As we show later, depending on the type action, the sets K_a and K_c are used as parameter K_s in $\mathcal{E}$, the encoding of kell containment conditions.

Formulas

$\mathcal{T}_p$, defined in Figure 6.12, is the encoding of kμ formulas into MMC formulas. The encoding follows from kμ semantics specified in Figure 4.4 (cf. Section 4.7.2).

Diamond and set modalities may receive an action condition or a set of action conditions. Action conditions, δ in kμ have the form (α, γ) or $(\alpha_\tau, \gamma, \gamma)$. α and α_τ identify the actions of interest; γ impose kell containment requirements on the location of the actions specified by α and α_τ.

In MMC it is only possible to identify actions of interest. Therefore the kμ encoding lifts the kell containment conditions from the transitions into the formulas being checked. Notice, in Figure 6.12, the γ_π, corresponding to the MMC encoding of kell containment conditions γ, occurs in the formulas following the diamond and box modalities.

Kell-m actions and kell containment conditions are translated into MMC by $\mathcal{AE}_p$. Kell containment sets are specified in the result of $\mathcal{A}_p$. The sets are used by $\mathcal{E}_p$ in the encoding of kell containment conditions. Name a is used to represent channel actions and name k is used to represent kell actions. When encoded in MMC, both a and k are represented as channel actions. Since MMC is first-order, name h is used as parameter in kell concretions to indicate a higher-order indicator (cf. Figure 6.4).

To prove the encoding is correct, it is necessary to demonstrate, for any kμ formula $\mathcal{F}$: (a) the result $\mathcal{T}_p(\mathcal{F})$ of the encoding is a MMC property; and (b) if $P \models_{\mathcal{V}} \mathcal{F}$ for an interpretation of formula parameters $\mathcal{V}$, then $\mathcal{I}[\![\ \emptyset, P\ ]\!] \models \mathcal{T}_p(\mathcal{F})$. (a) and (b) can be proved by structural induction.

We informally argue about the correctness of the encoding of $\langle\alpha, \gamma\rangle.\mathcal{F}$. A similar argument can be made for the other types of kμ formulas. Notice the actions returned by $\mathcal{A}_p$ are MMCπ actions, and therefore valid action specifications in diamond and box modalities within MMC properties. $\mathcal{E}_p$ returns either `tt` or a `pred` specification, both valid MMC properties. The MMC encoding of $\langle\alpha, \gamma\rangle.\mathcal{F}$ is, by definition:

$$\mathcal{T}_p(\langle\alpha, \gamma\rangle.\mathcal{F}) \stackrel{def}{=} \langle\alpha_\pi\rangle.(\gamma_\pi \wedge \mathcal{T}_p(\mathcal{F})), \text{ with } \mathcal{AE}((\alpha, \gamma)) = (\alpha_\pi, \gamma_\pi)$$

Assuming, by structural induction, that $\mathcal{T}_p(\mathcal{F})$ is a valid MMC property formula, then $\langle\alpha_\pi\rangle.(\gamma_\pi \wedge \mathcal{T}_p(\mathcal{F})$ is a valid MMC property formula.

According to kμ semantics, the meaning of $\langle\alpha, \gamma\rangle.\mathcal{F}$ is:

$$P \models_{\mathcal{V}} \langle\alpha, \gamma\rangle.\mathcal{F} \text{ when } \exists Q : P \xrightarrow{\alpha', \kappa} Q \ \wedge\ cmp(\alpha, \alpha') \ \wedge\ kc(\gamma, \kappa) \ \wedge\ Q \models_{\mathcal{V}} \mathcal{F}''$$

cmp, defined in Figure 4.6, decides if an action specification in a kμ formula matches an action in the extended LTS. *kc* decides if a kell containment condition holds for a given kell containment set. $\mathcal{F}''$ is $\mathcal{F}$ after alpha converting (replacing) any parameters in α with actual values used in the communication. In the MMC encoding we require a transition $P_\pi \xrightarrow{\alpha_\pi} Q_\pi$ after which both, the kell containment condition γ_π and the encoding of $\mathcal{F}$, must hold. P_π is the MMC process corresponding to P and Q_π to Q.

$$
\begin{array}{rcl}
\mathcal{T}_p(\mathtt{tt}) & \stackrel{def}{=} & \mathtt{tt} \\
\mathcal{T}_p(\mathtt{ff}) & \stackrel{def}{=} & \mathtt{ff} \\
\mathcal{T}_p(\neg\mathcal{F}) & \stackrel{def}{=} & \mathtt{neg\_form}(F), \text{ with } F \stackrel{def}{=} \mathcal{T}_p(\mathcal{F}) \\
\mathcal{T}_p(\mathcal{C}.\mathcal{F}) & \stackrel{def}{=} & \mathtt{pred}(\mathcal{C}, \mathcal{T}_p(\mathcal{F})) \\
\mathcal{T}_p(\mathcal{F}_1 \wedge \mathcal{F}_2) & \stackrel{def}{=} & \mathcal{T}_p(\mathcal{F}_1) \wedge \mathcal{T}_p(\mathcal{F}_2) \\
\mathcal{T}_p(\mathcal{F}_1 \vee \mathcal{F}_2) & \stackrel{def}{=} & \mathcal{T}_p(\mathcal{F}_1) \vee \mathcal{T}_p(\mathcal{F}_2) \\
\mathcal{T}_p(\langle\delta\rangle.\mathcal{F}) & \stackrel{def}{=} & \langle\alpha_\pi\rangle.(\gamma_\pi \wedge \mathcal{T}_p(\mathcal{F})), \text{ with } \mathcal{AE}(\delta) = (\alpha_\pi, \gamma_\pi) \\
\mathcal{T}_p(\langle-\delta\rangle.\mathcal{F}) & \stackrel{def}{=} & \langle-\alpha_\pi\rangle.\mathcal{T}_p(\mathcal{F}) \vee \langle\alpha_\pi\rangle.(\mathcal{T}_p(\mathcal{F}) \wedge \neg\gamma_\pi), \text{ with } \mathcal{AE}(\delta) = (\alpha_\pi, \gamma_\pi) \\
\mathcal{T}_p(\langle S\rangle.\mathcal{F}) & \stackrel{def}{=} & \left\{\begin{array}{l} \mathtt{ff}, \text{ if } S = \emptyset; \text{ otherwise:} \\ \mathcal{T}_p(\langle\delta_1\rangle.\mathcal{T}_p(\mathcal{F})) \vee \mathcal{T}_p(\langle\delta_2\rangle.\mathcal{T}_p(\mathcal{F})) \vee \cdots \vee \mathcal{T}_p(\langle\delta_n\rangle.\mathcal{T}_p(\mathcal{F})), \\ \text{with } S = \{\delta_1, \delta_2, ..., \delta_n\} \end{array}\right. \\
\mathcal{T}_p(\langle\text{-S}\rangle.\mathcal{F}) & \stackrel{def}{=} & \left\{\begin{array}{l} \langle-\rangle.\mathcal{T}_p(\mathcal{F}), \text{ if } S = \emptyset; \text{ otherwise:} \\ \mathcal{T}_p(\langle-\delta_1\rangle.\mathcal{T}_p(\mathcal{F})) \wedge \mathcal{T}_p(\langle-\delta_2\rangle.\mathcal{T}_p(\mathcal{F})) \wedge \cdots \wedge \mathcal{T}_p(\langle-\delta_n\rangle.\mathcal{T}_p(\mathcal{F})), \\ \text{with } S = \{\delta_1, \delta_2, ..., \delta_n\} \end{array}\right. \\
\mathcal{T}_p([\delta].\mathcal{F}) & \stackrel{def}{=} & \mathtt{neg\_form}(F), \text{ with } F \stackrel{def}{=} \langle\alpha_\pi\rangle.(\gamma_\pi \wedge \mathtt{neg\_form}(F')), \\
 & & F' \stackrel{def}{=} \mathcal{T}_p(\mathcal{F}), \mathcal{AE}(\delta) = (\alpha_\pi, \gamma_\pi) \\
\mathcal{T}_p([-\delta].\mathcal{F}) & \stackrel{def}{=} & \mathcal{T}_p([-\{\delta\}].\mathcal{F}) \\
\mathcal{T}_p([S].\mathcal{F}) & \stackrel{def}{=} & \left\{\begin{array}{l} \mathtt{tt}, \text{ if } S = \emptyset; \text{ otherwise:} \\ \mathcal{T}_p([\delta_1].\mathcal{T}_p(\mathcal{F})) \wedge \mathcal{T}_p([\delta_2].\mathcal{T}_p(\mathcal{F})) \wedge \cdots \wedge \mathcal{T}_p([\delta_n].\mathcal{T}_p(\mathcal{F})), \\ \text{with } S = \{\delta_1, \delta_2, ..., \delta_n\} \end{array}\right. \\
\mathcal{T}_p([\text{-S}].\mathcal{F}) & \stackrel{def}{=} & \left\{\begin{array}{l} [-].\mathcal{T}_p(\mathcal{F}), \text{ if } S = \emptyset; \text{ otherwise:} \\ \mathcal{T}_p([-\delta_1].\mathcal{T}_p(\mathcal{F})) \wedge \mathcal{T}_p([-\delta_2].\mathcal{T}_p(\mathcal{F})) \wedge \cdots \wedge \mathcal{T}_p([-\delta_n].\mathcal{T}_p(\mathcal{F})), \\ \text{with } S = \{\delta_1, \delta_2, ..., \delta_n\} \end{array}\right. \\
\mathcal{T}_p(F(\widetilde{p})) & \stackrel{def}{=} & \mathtt{form}(F'), \text{ with } F' \stackrel{def}{=} \mathcal{T}_p(\mathcal{F}_d\{\widetilde{p}/\widetilde{c}\}), \text{ having } F(\widetilde{c}) \stackrel{def}{=} \mathcal{F}_d
\end{array}
$$

Where,

$$
\mathcal{AE}_p(\delta) \stackrel{def}{=} \left\{\begin{array}{l}
(\mathcal{A}_p(\alpha), \mathcal{E}_p(\gamma, K_c)), \text{ if } \delta \equiv (\alpha, \gamma) \text{ and } \left\{\begin{array}{l} \mathcal{A}_p(\alpha) = \overline{a}(\widetilde{w}, K_a, K_c, B) \\ \text{or} \\ \mathcal{A}_p(\alpha) = \overline{k}(h, K_a, K_c, B) \end{array}\right. \\
(\mathcal{A}_p(\alpha), \mathcal{E}_p(\gamma, K_a)), \text{ if } \delta \equiv (\alpha, \gamma) \text{ and } \left\{\begin{array}{l} \mathcal{A}_p(\alpha) = a(\widetilde{c}, K_a, K_c, B) \\ \text{or} \\ \mathcal{A}_p(\alpha) = k(x, K_a, K_c, B) \end{array}\right. \\
(\mathcal{A}_p(\alpha_\tau), \mathcal{E}_p(\gamma_a, K_a) \wedge \mathcal{E}_p(\gamma_c, K_c)), \\
\qquad \text{if } \delta \equiv (\alpha_\tau, \gamma_a, \gamma_c) \text{ and } \left\{\begin{array}{l} \mathcal{A}_p(\alpha_\tau) = \tau(\overline{a}(\widetilde{w}, K_a, K_c, B)) \\ \text{or} \\ \mathcal{A}_p(\alpha_\tau) = \tau(\overline{k}(h, K_a, K_c, B)) \end{array}\right.
\end{array}\right.
$$

Figure 6.12: Encoding of kμ Formulas in MMC

Let us assume the formula holds in kell-m but not in MMC. This means there is no transition $P_\pi \xrightarrow{\alpha_\pi} Q_\pi$ after which $Q_\pi \not\models (\gamma_\pi \wedge \mathcal{F}_p)$. This may happen only if there is no $P_\pi \xrightarrow{\alpha_\pi} Q_\pi$, or if $(\gamma_\pi \wedge \mathcal{F}_p)$ does not hold at Q_π.

Because a kell-m process and its MMC encoding are behaviourally equivalent (Appendix B.2), such a transition $P_\pi \xrightarrow{\alpha_\pi} Q_\pi$ must exist. Therefore the only possibility is $(\gamma_\pi \wedge \mathcal{F}_p)$ does not hold at Q_π. This could happen if γ_π does not hold at Q_π, or if $\mathcal{F}_p$ does not hold at Q_π. Let us assume γ_π does not hold at Q_π. This implies the kell containment set, built for the action in the encoding, does not match the kell containment set as specified in the extended LTS semantics for P.

For kell abstractions and channel abstractions and concretions, $\mathcal{A}$ builds the kell containment set. For kell concretions $\mathcal{S}$ builds the kell containment set. Every time an action is lifted from a kell, $\mathcal{A}$ adds the action to the associated kell containment set (cf. Figure 6.7). When the passivation code for a kell is generated (cf. Figure 6.5), the kell containment set for the kell concretion is updated with kell information as passivation code is generated from the external kells to the nested kells. These kells are included as parameters in the actions when the actions are encoded in MMC.

Assuming the kell containment sets are properly built in the encoding, and also assuming the kell containment sets are available after the actions, the kell containment condition may still not hold if it is not properly encoded by $\mathcal{E}_p$. Since $\mathcal{E}_p$ implements *kc* of the kμ kell containment semantics (cf. Figure 4.6), the kell containment condition must hold. Consequently, the only remaining case is $\mathcal{F}_p$ not holding at Q_π. But we argue, by structural induction, if $\mathcal{F}$ is not a diamond modality, $Q_\pi \models \mathcal{F}_p$ and $P \models_{\mathcal{V}} \langle\alpha,\gamma\rangle.\mathcal{F} \Rightarrow P_\pi \models \mathcal{T}_p(\langle\alpha,\gamma\rangle.\mathcal{F})$. If $\mathcal{F}$ is a diamond modality, we apply the argument above as many times as necessary until the unfolding of $\mathcal{F}$ leads to a non-diamond formula, at which point we can argue by structural induction.

Conversely, $\mathcal{T}_p(\langle\alpha,\gamma\rangle.\mathcal{F})$ may hold for a process P_π, but $P \not\models_{\mathcal{V}} \langle\alpha,\gamma\rangle.\mathcal{F}$, where P is the kell-m process corresponding to P_π. Because of the behavioural equivalence of kell-m processes and their MMC encoding, this can only happen if the kell containment condition holds for a transition in the MMC process but not in the kell-m process, or if $\mathcal{F}$ holds after the transition in the MMC process but not in the kell-m process. Assuming kell containment sets and conditions are properly translated, if $\mathcal{F}$ is not a diamond modality we can argue, by structural induction, this cannot occur and $\mathcal{T}_p(\langle\alpha,\gamma\rangle.\mathcal{F}) \Rightarrow \langle\alpha,\gamma\rangle.\mathcal{F}$. If $\mathcal{F}$ is a diamond modality we need to, once again, unfold $\mathcal{F}$ until we get to a non-diamond formula. During the unfolding of diamond formulas we argue the only way for the unfolded formula not to apply is for the subformula, after the diamond modality, not to apply.

The encoding of $\langle -(\alpha,\gamma)\rangle.\mathcal{F}$ holds if there is a transition with action different than α_π after which the encoding of $\mathcal{F}$ holds, or if there is at least one transition with action α_π but γ_π does not hold and the encoding of $\mathcal{F}$ does. α_π is the MMC action corresponding to

kell-m action α, and γ_π is the MMC kell containment condition corresponding to kell-m's γ. Notice the encoding implements the semantics of $\langle -(\alpha, \gamma)\rangle.\mathcal{F}$ as defined in Figure 4.4.

We use a well known modality equivalence for the encoding of $[\delta].\mathcal{F}$ [50]: $[\delta].\mathcal{F} \equiv \neg(\langle\delta\rangle.\neg\mathcal{F})$. The encoding of the other formulas follow from Figure 4.4.

6.4 Tool Application and Performance

The complexity of model checking a kell-m process depends on the size of the LTS for the process and the structure (i.e. recursion nesting) of the property being verified (cf. Section 4.6). Our tool operates on the LTS obtained using the extended kell-m semantics (cf. Section 3.5.3).

Within our tool the LTS for a process is obtained by means of MMC's Prolog predicate `trans` (cf. Section 6.3.2). Given a node in the LTS, the `trans` predicate computes the immediate transitions for the node. The LTS is generated, on demand, as required by the verification. As previously mentioned in Section 6.3.1, when executing using XSB Prolog [11], the `trans` predicate is tabled, meaning XSB caches the result of evaluating the predicate. Hence, future evaluations of the predicate are not re-evaluated if the results have already being tabled. For some predicates, tabled evaluation provides a performance advantage of an order of magnitude over the non-tabled predicates, although it increases memory requirements [143, 137].

The experiments were performed on a server with two quad-core Intel® Xeon® E5430 2.66GHz CPUs and 16GBs of RAM running Linux 2.6.27-14 64 bits. In the experiments memory usage was restricted to 8GBs.

Due to the increased memory requirements of tabled evaluation in XSB, two ports of the kell-m checker were implemented. One port was developed using XSB Prolog, the other port was developed using SWI Prolog [168]. SWI is a Prolog interpreter without tabled evaluation. The port to use can be specified when invoking the hl-kell-m checker. For the SWI port there is the option of pre-computing the LTS previous to the verification of properties. This option is useful when verifying multiple properties for the same model. If the option is not used the LTS is generated as the verification requests are executed.

Although the server on which the experiments were performed has 8 cores, neither XSB nor SWI Prolog are capable of using more than one core at a time. Hence, performance on servers with fewer cores is comparable. This observation was confirmed by performing the experiments on a computer with a two-core Intel® Core® 2 Duo P8400 2.26GHz CPU and 3 GBs of RAM running Linux 2.6.30 64 bits.

We start the evaluation of the performance of our tool by verifying several of the properties defined in Section 5.2 for the Core (i.e. *coreapi_debs*) and Optional (i.e. *optapi_debs*)

Property	Type	Verif. Result	Core API			Optional API		
			XSB	SWI	SWI-L	XSB	SWI	SWI-L
of_interest_only	safety	yes	50.279	26.820	5.770	*	1137.460	36.460
delivery_before_pub	safety	no	0.116	1.050	0.300	14.708	36.210	1.800
delivery_before_subsc	safety	no	0.020	0.350	0.100	0.084	3.030	0.370
all_notified	liveness	yes	133.100	2.390	1.020	*	365.880	20.530
matches_adv	safety	yes	n/a	n/a	n/a	*	3.190	0.950
from_site_only	locality	yes	33.708	21.340	3.640	*	68.030	5.880
service_migration	passivation	yes	506.15	15.530	3.350	*	35.190	4.380

Table 6.5: Verification CPU Time (in sec.) for Correct Core and Optional API Models

API specifications. Recall the properties were specified using reduction semantics.

In Table 6.5 we report the CPU time in seconds for the verifications. Column XSB corresponds to the port of the kell-m checker on XSB Prolog with tabled evaluation. XSB was invoked separately for each property verified. An asterisk (*) is used in the Table 6.5 to indicate no result was obtained because the maximum allocated memory was exceeded during the verification.

Column SWI corresponds to the SWI Prolog invoked separately for each property. Column SWI-L corresponds to the SWI Prolog port with construction of the LTS prior to the verification of the properties. With SWI-L all properties are verified with a single invocation of SWL Prolog. The time to construct the LTS in SWI-L was 12.670 seconds for *coreapi_debs* and 115.770 seconds for *optional_debs*. The LTS for *coreapi_debs* has 117 states and 224 transitions; for *optapi_debs* the LTS has 173 states and 361 transitions.

Safety property *matches_adv* in Table 6.5 applies to the Optional API only; it requires published events to match their advertisements (cf. Section 5.2.6).

The performance of the SWI port was better than the XSB port for all properties but *delivery_before_pub* and *delivery_before_subsc*. This result was surprising since we were expecting the XSB port to perform better than the SWI port for all properties. Upon inspection of the MMC code we found out the observed performance was due to the tabling of the `mck` predicate used in the verification. Recall predicate `mck(P, C)` is used in MMC to check if a condition `C` is satisfied by process `P` (cf. Section 6.3.2). It turns out when tabling is done there is no short-circuit evaluation for Prolog inferences. Consider a predicate `p :- a; b.`, meaning *infer* `p` *if* `a` *or* `b` *can be inferred*. When such a predicate is tabled, XSB evaluates both predicates `a` and `b`, even if `a`. This allows XSB to later infer `p` quickly if not `a`. The effect in the checker is such that when verifying a property $\mathcal{F}_1 \vee \mathcal{F}_2$, both $\mathcal{F}_1$ and $\mathcal{F}_2$ are verified in XSB even if it can be quickly decided the process being verified satisfies $\mathcal{F}_1$. For property *all_notified*, the performance of XSB without tabling predicate `mck` is 0.232 seconds for the Core API. Compare with 133.100 seconds when tabling is done.

Property	Type	Verif. Result	Core API (Faulty)			Optional API (Faulty)		
			XSB	SWI	SWI-L	XSB	SWI	SWI-L
of_interest_only	safety	no	*	94.580	41.590	*	797.300	48.280
delivery_before_pub	safety	yes	0.000	0.000	0.010	101.926	0.750	0.340
delivery_before_subsc	safety	yes	0.000	0.000	0.010	0.144	0.750	0.340
all_notified	liveness	no	222.083	58.160	12.540	*	363.970	22.400
matches_adv	safety	no	n/a	n/a	n/a	*	3167.480	222.570
from_site_only	locality	no	28.169	0.080	0.050	*	0.460	0.100
service_migration	passivation	no	0.884	5.770	1.240	*	31.170	3.840

Table 6.6: Verification CPU Time (in sec.) for Incorrect Core and Optional API Models

Tabling evaluation in XSB is also the cause behind the increased memory usage during the verifications when the XSB port of our tool was used. For example, XSB exceeds the memory allocation of 8GB for property *matches_adv* for the Optional API (cf. Table 6.5). Without tabling predicate `mck`, the property is verified in 0.160 seconds without exceeding the maximum memory allocation. Because the transition relation `trans` is also tabled, even without tabling `mck` the other properties in the table marked with an asterisk for XSB still exceeded the maximum memory allocation.

CPU times for the verifications when the models do not satisfy the properties are shown in Table 6.6. For this experiment both the *coreapi_debs* and *optapi_debs* models were altered to deliver notifications for uninterested subscribers and to skip notification for interested subscribers. We name the incorrect models *coreapi_debs_inc* and *optapi_debs_inc*. The time to construct the LTS in SWI-L for this experiment was 67.580 for *coreapi_debs_inc* and 244.970 seconds for *optapi_debs_inc*. The LTS for *coreapi_debs_inc* has 224 states and 544 transitions; for *optapi_debs* it has 314 states and 791 transitions.

Verification times are reported as 0.000 in Table 6.6 when the execution time is less than the resolution of the CPU timing feature in the Prolog implementations. When compared to the correct models, verification improved for the properties for which exceptions could be quickly found. For example, in the faulty models an event is delivered before publication or subscription. For property *matches_adv* the time in the faulty models represents the search on the whole LTS for an nonexistent advertisement matching a published event.

From the previous experiments we conclude the SWI ports are better suited for the verification of the kμ properties considered. In particular, tabling in the XSB port has an unfavorable impact on the performance of the verification.

Pre-computation of the LTS in the SWI-L port always improved the performance of the verification with the exception of the verification of the *delivery_before_pub* and *delivery_before_subsc* properties for the faulty models. And in these cases the benefit of not computing the LTS was of only 0.010 seconds.

Chain Depth	LTS Semantics States	Trans.	CPU Time (secs)	Reduction Semantics States	Trans.	CPU Time (secs)
1	6	8	0.000	2	1	0.000
2	18	38	0.010	3	2	0.000
3	54	156	0.020	4	3	0.000
4	162	594	0.170	5	4	0.000
5	486	2160	1.430	6	5	0.000
6	1458	7614	12.360	7	6	0.000
7	4374	26244	108.430	8	7	0.010
8	*	*	971.910	9	10	0.010

Table 6.7: Performance for Verification on Chains of Communications

We continue the evaluation of the performance of our tool by modelling a simple process communicating on channel a_0:

$$\textbf{process } c_1() \ \{ \ c_0() \mid a_0(X) \triangleright \overline{a_1}(X) \ \}$$

with c_0:

$$\textbf{process } c_0() \ \{ \ \overline{a_0}(\text{“}msg\text{”}) \ \}$$

When c_1 is composed with process c_2 below, the message is read on channel a_1 and written on channel a_2:

$$\textbf{process } c_2() \ \{ \ c_1() \mid a_1(X) \triangleright \overline{a_2}(X) \ \}$$

In c_2 we say there is a chain of communication depth two, since two τ actions, one after the other occur. In Table 6.7 we report the CPU time in the number of seconds it takes to verify a property $\mathbf{E}_e(\langle \overline{b}(X) \rangle)$ by the SWI port of the kell-m checker using LTS and reduction semantics on processes with different chain depths d. We define c_d as:

$$\textbf{process } c_d() \ \{ \ c_{d-1}() \mid a_{d-1}(X) \triangleright \overline{a_d}(X) \ \}$$

Notice the property is not satisfied since all actions occur in channels a_i while the property requires a write on channel b. Such property typically requires the tool to walk the whole LTS. CPU times of 0.000 in Table 6.7 for reduction semantics, as previously mentioned, indicate the execution took less than the resolution of the CPU timing feature in SWI. An asterisk is used to indicate that the number of states and transitions was not computed because the program ran out of memory.

Verification time for LTS semantics increases by an order of magnitude for each chain-depth. The results of this experiment indicate our tool is not well suited for verification

Kell Nesting Depth	LTS Semantics			Reduction Semantics		
	States	Trans.	CPU Time (secs)	States	Trans.	CPU Time (secs)
6	83	194	0.010	2	1	0.000
7	163	386	0.050	2	1	0.000
8	323	770	0.200	2	1	0.000
9	643	1538	0.820	2	1	0.000
10	1283	3074	3.720	2	1	0.000
11	2563	6146	15.830	2	1	0.000
12	5123	12290	70.400	2	1	0.000

Table 6.8: Performance for Verification on Nested Kells

using LTS semantics where there are communication chains of length 8 or greater. However, this performance degradation is not observed when using reduction semantics.

The fact the maximum allowed memory was reached when computing the LTS size for a chain-depth of 8 is indication the LTS is not stored efficiently in memory. Currently the LTS is stored in SWI as facts `(SourceProcess, TransitionAction, DestinationProcess)`.

We now measure the effect of nested kells on the verification. Specifically we look at processes:

$$K_n[K_{n-1}[\cdots K_0[\overline{a}(c)]]] \mid K_0[X] \triangleright X$$

We verify property $\mathbf{E}_e(\langle \overline{T}[Y] \rangle)$ for LTS semantics and $\mathbf{E}_e(\langle \overleftrightarrow{T}[Y] \rangle)$ for reduction semantics. Neither of these properties is satisfied by the process and the verification needs to generate the whole LTS. CPU time in seconds is reported in Table 6.8. The resolution of the CPU timing feature in Prolog is not high enough to measure the execution time for kell nesting of depth less than 6.

Kell nesting has no impact on the size of the LTS when verification of the sample process is done using reduction semantics. When using LTS semantics the performance of the tool quickly degrades for kell nesting of depth greater than 10.

Although model checking of the *coreapi_debs* and *optapi_debs* models was successful with the SWI and SWI-L ports, the results in this section indicate the tool is not well suited for larger models and further work is required to improve the performance of the verification. The tool nevertheless confirms the feasibility of model checking kell-m processes.

As proposed in Future Work (cf. Section 7.1), there are two main areas where the tool can be improved. The first area of improvement is in the generation of the LTSs. When reduction semantics, the resulting LTSs for the evaluated models are small with

only hundreds of nodes and transitions. Even though the LTSs are small, the tool takes 115.770 seconds to generate the LTS for the *optapi_debs* model.

The other area of improvement is in the reduction of the size of the LTSs. This is particularly important for LTS semantics where the LTSs can be considerably bigger than the LTSs obtained with reduction semantics (e.g., Table 6.7). For example, although $P|\mathbf{0}$ and $\mathbf{0}|P$ are structurally equivalent, there is one node for each of these processes in the LTS produced by our tool. The implication of this redundancy is that every transition from $P|\mathbf{0}$ is also a transition from $\mathbf{0}|P$. The size of the LTS can be reduced by having only one node in the LTS for each structurally equivalent process derived during the evolution of the system being verified. This approach has been used successfully in model checkers for other process algebras ([e.g., 170]). A related improvement is an efficient representation of the LTS in memory. Currently the LTS is stored as prolog facts.

6.5 Related Work

Our kell-m checker is the only tool, we are aware of, for model checking kell-based process algebras. In Sections 3.6 and 4.7 we have already presented other tools for model checking other process algebras, including the Mobility Workbench [160, 161], the Mobility Model Checker [171], and Logic for Mobile Ambients [147].

Ferrari et al. developed HAL, an automata-based tool for verification of systems specified in the π-calculus [66]. π-logic, a temporal logic with modalities, is used for property specification. π-calculus processes are transformed into automata, and π-logic formulas are translated into action computation tree logic (ACTL) formulas [124]. ACTL has the same expressive power as CTL* [59]. There is no support in the tool for higher-order expressions, and the π-logic does not support fixed-points nor recursive formulas.

Traditional model checkers have been used to encode process algebras. Song and Compton used SPIN to verify monadic π-calculus processes [154]. Monadic processes can only transmit one name as part of communications. The logic supported in SPIN for property specification is linear temporal logic LTL. Being the μ-calculus a branch logic, there is no way to automate the encoding of μ-calculus properties into LTL properties. Hence, properties can only be specified on the translated π-calculus processes and not in the original π-calculus specification. No support is provided in this work for higher-order π-calculus.

6.6 Summary and Contributions

We described our implementation of a prototype tool for model checking systems represented using kell-m. The tool receives as input process specifications in hl-kell-m and

property specifications in hl-kμ, along with verification requests. Verification requests have the form **check** *property* **for** *process*. The output of the tool is a report indicating, for each verification requests, whether the kell-m process satisfies the property or not.

Our tool implements two parsers. One parser is used to transform hl-kell-m processes to kell-m processes. The other parser is used to transform hl-kμ to kμ. The tool has the option of displaying the kell-m process using the same symbols used in the description of kell-m in Chapter 3.

Our tool itself is implemented using an extended version of the Mobility Model Checker (MMC). MMC is a tool for the verification of processes specified using MMC's variation of the first-order synchronous π-calculus. MMC itself is implemented using Prolog. Our tool encodes kell-m processes as MMC π-calculus processes. A callback feature in MMC allows us to deal with higher-order expressions in kell-m. kμ formulas are encoded using MMC's formalisms for property specification. The model checking features in MMC are used to verify the translated processes and properties.

Two ports of our tool were implemented. One port uses XSB Prolog, a Prolog implementation with tabled evaluation. The other port uses SWI Prolog, a regular implementation of Prolog. With the tool we verified two of the models presented in Chapter 5. Based on the performance exhibited by our tool during the verifications we concluded further research is needed to be able to handle larger models.

To the best of our knowledge, the use of our tool to model check the models of Chapter 5, has been the first time safety and liveness properties previously identified for DEBSs have been verified. It has also been the first time a Kell-based process has been model checked, and the first time locality of actions has been verified for a process algebra derived from the π-calculus.

Chapter 7

Conclusion and Future Work

In this thesis we developed formalisms and models supporting the specification, prediction, and verification of behaviour in DEBS middleware and application components interacting via events. Our approach and contributions are summarized as follows.

Event Model Categorization. Initially, we compared and categorized the event models found in DEBSs with the event models in other IISs. The event model determines the event related features in a system including the announcement, subscription, and delivery of events. With this categorization we elicited the key system behaviour to be supported by the formalisms and models. We expect this categorization to be of interest in areas related to the development of design and structuring abstractions for DEBSs.

Generic DEBS Models. Based on the Common API, a proposal for a standard DEBS API, we developed generic models formalizing the event-related features previously identified. Most DEBSs readily support the Common API or can be easily modified to support it. The models we developed are compositional and parameterize the features typically varying among different DEBS implementations such as filtering capabilities, event delivery semantics, and support for structured events. Models for specific systems are obtained by composing models representing how the parameterized features are provided by the DEBS of interest with a generic model where these features are parameterized.

Properties previously identified for DEBSs were specified based on our models. Using the reasoning capabilities of the formalisms developed in this thesis it is possible, for the first time, to verify these properties. We showed how new properties, not expressible with the previous work in the area, can now be specified in our work. These properties impose conditions on the locations of actions performed in the DEBSs, on the adaptation of functionality by publisher and subscriber components, and on the migration of functionality.

Models for Specific DEBSs. We modelled REBECA, a research DEBS supporting a novel structuring mechanism for the restriction of event visibility. In REBECA it is possible to define groups of publisher and subscriber components. These groups of components, called scopes, can also include other groups. Subscribers are notified of events of interest only if the publisher and subscriber components are within the same scope. We derived the model for REBECA from the Common API model and showed how safety and liveness properties previously identified by others for scopes are specified for the model of REBECA. Similar to the properties for regular DEBS, this is the first time the properties for scoped DEBSs can be verified. Our model of REBECA showcases the ability to model an incoming DEBS using the formalisms we proposed in this thesis.

The main concern of the previous models was the representation of event-related behaviour in DEBSs. Specifically, the behaviour exhibited by the DEBS as events are subscribed to, published, and notified. With a model of NaradaBrokering we represented the physical structure of brokers in this DEBS. Brokers are administrative components in charge of the communication between the middleware and the publisher and subscriber components. In NaradaBrokering brokers are hierarchically structured forming a broker network. This model showcases the use of our work to model a non event-related facet of a DEBS. In particular, the model allows for the specification of properties related to the adaptation of the structure and the brokers within the broker network.

Formalisms for System and Property Representation. All the models in this thesis were developed using kell-m, our new asynchronous process algebra with hierarchical localities. Properties for the systems modelled using kell-m were specified using kμ, our new modal temporal logic. kμ is the first property specification formalism proposed for a kell-based algebra.

Process algebras are formalisms for the study of behaviour of concurrent systems. In kell-m systems are represented as processes executing in parallel. The processes can execute within localities called kells, and kells can be within other kells. Kell-m is based on the kell-calculus family of process algebras. In contrast with these algebras, in kell-m communication is not restricted to processes within a location. The mode of communication in kell-m closely resembles the communication in DEBSs where components can communicate irrespectively of their locations.

Besides the rules governing communication, we illustrated the advantages of kell-m over other process algebras based on kell-m's support for higher-order expressions, localization of communication actions, process passivation, process migration, and model checking.

The ability to localize actions in kell-m is due to kell-m's novel operational semantics. Traditionally, the operational semantics for process algebras only specify the rules governing communication actions among the processes. In kell-m, operational semantics

are extended with information about the kells where the actions are taking place. This extension introduces location-aware reasoning capabilities to the algebra and allows the specification of DEBS behaviour that can be restricted to specific locations. We expect this feature to be useful in other areas of software engineering where it is required to formalize the locations where behaviours of interest occur.

Process passivation is the mechanism that allows a process in kell-m to stop or alter the process within a kell. This mechanism is available in all kell-based process algebras. Process passivation can be used to model situations in which DEBS components adapt their behaviour by themselves as well as situations in which DEBS component behaviour is adapted by other components. With our work, for the first time for kell-based algebras, it is possible to specify the conditions under which such adaptation takes place.

Using process passivation, kells and the processes executing in them can be moved to other kells. With our work, the conditions under which such process migration takes place can be formalized. For example, a component providing a service may migrate to another computer with more resources once it is notified of an event indicating that the resources at its current location are running low. We expect this feature to be useful in non-DEBS areas where there is a need to formalize the conditions under which mobility takes place.

Languages and Prototype Tool. To improve the readability of the models and property specifications we introduced hl-kell-m and hl-kμ, sugared versions of kell-m and kμ. We illustrated how traditional control and modularization constructs can be represented using kell-m. We provided grammars for hl-kell-m and hl-kμ and implemented parsers for the translation to kell-m and kμ. These parsers are part of a prototype model checking tool encoding the operational semantics of kell-m. This tool receives as input models in hl-kell-m, property specification in hl-kμ, and verification requests. The output of the tool is a report indicating for each verification request the outcome of the verification.

Two ports of the model checking tool were implemented, one port uses XSB Prolog and the other uses SWI Prolog. Performance of the tool on each one of the ports was evaluated using the models and several of the properties previously proposed for the Common API. From the evaluation we concluded further work is required to handle larger models. However the tool illustrates the feasibility of model checking kell-m models representing DEBSs. Using the tool, for the first time liveness and safety properties previously proposed in the area for DEBSs were successfully verified on DEBS models.

Other Contributions. As part of the development of the formalisms and tools we have new results in the area of kell-based process algebras. These contributions are not directly related to our goal of specifying, predicting and verifying the behaviour in DEBSs and are described below.

We defined semantics of kell passivation in terms of higher-order channel communications. We showed that a kell can be modelled using higher-order π-calculus as a process that, non-deterministically, can be passivated or can advance in its execution. This result allows for an encoding of kell-m and other kell-based process algebras into higher-order π-calculus.

We also defined bisimilarity up to kell containment, a new kind of bisimilarity for kell based process algebras. Behavioural equivalences allow us to determine if two processes have the same behaviour based on certain criteria. In the case of bisimilarity up to kell containment, two processes have the same behaviour if their location structure is the same. Without this bisimilarity, checking if two processes have the same locality structure requires the computation of a stronger form of bisimilarity called bisimulation congruence. The usefulness of the result lies in the ability to check if two models have the same kell structure.

Finally, the traditional approach in process algebras for reducing higher-order expressions into first-order expressions was developed by Davide Sangiorgi in his PhD thesis [145]. We showed the approach is not applicable to kell-m nor kell-based process algebras. The implication is that there is no straight forward encoding of kell-m into first-order π-calculus. The intuition is that in the π-calculus, higher-order can be represented as remote code invocation, while in kell-m higher-order expressions involving kells require code shipping and local execution.

7.1 Future Work

We envision several directions of research based on the models we developed, the formalisms we proposed, and the tools we implemented.

Component Relationships. To continue improving the understanding of DEBS and applications it is necessary to study the relationships arising among components in DEBSs. In general, components can generate, forward, filter, transform, and consume events. When an event is transformed the transformation can be one of translation, aggregation, splitting, or enrichment [60]. Translation occurs when the data in the event is modified to another representation. Semantically, the translated data may or may not be equivalent to the original data. Aggregation represents the case when data from more than one event is aggregated and published as another event. Splitting occurs when data from one event is re-published as several events. Enrichment occurs when the data from the received event is complemented with data from other sources and republished as a different event.

Other kinds of relationships may occur among components as they execute application level functionality. The simplest and most common relationship is the *react* relationship,

where a subscriber component alters its behaviour upon notification of an event. A component may *delegate* the handling of an event to another component. Yet another possible relationship between components may occur when the functionality of a component C_e *extends* the functionality of another component C. C_e is interested in at least the same events C is interested, and C_e publishes at least the same events C publishes. Moreover, the functionality executed in C as a reaction to an event must also be observable in C_e. Similarly, the functionality that causes C to publish an event must also be observable in C_e.

Our models and formalisms can be used in the formalization and study of these component relationships. We consider the study of these relationships key in better understanding how functionality is composed in DEBS applications. Ultimately such understanding can be helpful in the creation of structuring, information hiding, and modularization abstractions for DEBSs.

Context-aware Applications. The ability to support context-aware applications is a requirement that, we believe, will greatly influence the DEBS event model. Dey defines context as: any information that can be used to characterize the situation of an entity [53]. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves.

Context-aware applications are able to alter their behaviour based on patterns of use, location, and timing conditions. In our categorization of event models, we included context in the filtering of events. But context-awareness has the potential of affecting many more aspects of the event model. For example, consider a component that subscribes to events when it is within a certain geographical region, and that unsubscribes when it leaves the region. In fashion with adaptive applications, the subscribe and unsubscribe operations should be implicitly invoked on behalf of the component without the need for the component to be constantly checking its location coordinates. Context-awareness also influences publishers of events. Events may be generated because of particular conditions in the context of the component. This context may in turn be associated to the published event and be transmitted along with the event attributes to interested subscribers. Contextual information for the event itself (e.g., the time the event was generated), may be of interest as well. Research into the development of context-aware applications is very active, including the development of context-models for representing contextual information [24]. Since context-aware applications have been proposed for the same type of applications as DEBSs, we believe that the research areas will eventually intersect. As a first step, we are currently specifying situations in which context-awareness affects the DEBS event model and the interactions between publisher and subscriber components. We envision using the location-awareness features in kell-m in the specifications and are also exploring time-awareness extensions to kell-m.

Typed Communications. Besides the time-awareness extension previously mentioned, future work for kell-m includes the addition of typed communications, support for synchronous interactions, and operational semantics exposing structural information about the processes.

With typed communications matching between abstractions and concretions will occur not only if the channels have the same name, but also if the types of the parameters output in the concretion match the types of the parameters expected in the abstraction. We expect the introduction of typed communications will improve the readability of the models and will help in the early identification of simple but frequently occurring errors in the models. For example, the process modelling publication of events in the Optional API expects an advertisement as its first parameter and the event as its second parameter (cf. Section 5.2.6). Currently, when these parameters are swapped in a model, this error is typically only caught until a property related to the publication of events is being verified for the model. With typed communications, the error can be found and corrected by the tools before any property is even specified.

Synchronous Interactions. Systems being modelled may combine synchronous and asynchronous interactions. An example of a synchronous interaction is a remote procedure call where the caller is blocked until the remote procedure executes. An example of an asynchronous interaction is the publication of an event and the reaction to the event by a subscribed component. Being an asynchronous process algebra, kell-m is well suited for the representation of asynchronous interactions such as the ones occurring DEBSs. Specifically, abstractions in kell-m are blocking and concretions are non-blocking. Without using return channels, there is no way in kell-m for a process writing on a channel to know when the reading process receives the data sent in the communication. Hence, currently in kell-m synchronous interactions are represented as follows:

$$\overline{a}(\mathit{data},\ \mathit{rc}) \mid \mathit{rc}() \triangleright P$$

In the previous process, P is executed only until the data sent on channel a is received by a process and the receiving process acknowledges the reception of data by using channel rc:

$$a(\widetilde{d}, \mathit{rc}) \triangleright (\overline{\mathit{rc}}() | Q)$$

By extending kell-m with synchronous communications, we could write instead:

$$\overline{a}(\mathit{data}).P \mid a(\widetilde{d}) \triangleright Q$$

The main benefit of this change is in the simplification of the models for systems with synchronous interactions.

Observability of Higher-Order Communications. The logic for mobile logics, proposed for the ambient calculus (cf. Section 4.7.4) allows the specification of properties such as *process $P|Q$ executes within ambient a*. Ambients in the ambient calculus are the equivalent of kells in kell-m. These properties are not expressible in our work because kμ is based on the kell-m assumption that only communication actions are observable, as proposed by Sangiorgi and Milner [117].

Properties specifying the structure of a process can be of interest when modelling process adaptation. Currently in kell-m, it is possible to specify an adaptation is taking place as well as the the behaviour of the adapted process. By supporting the ambient style of properties we could also allow the specification of the exact structure the process must have after an adaptation. As we extended the kell-m semantics to make kell containment observable, in theory, we could extend the semantics even further to make process composition and null process observable.

Property Patterns. Dwyer et al. surveyed 500 examples of property specifications for finite-state verification tools, and found that 92% of the surveyed properties were instances of a group of patterns they have documented [56]. The property patterns proposed by Dwyer et al. are parameterizable, high-level, formalism-independent specification abstractions. We plan to specify, for each kind of pattern, selected properties using kμ. We are looking for property patterns commonly occurring within the context of DEBSs.

Tools. To improve the performance and scalability of our tool we plan to encode kell-m operational semantics without translation to MMC. The main requirements for the encoding are to speed up the generation of the LTS and to have an efficient representation of the LTS in memory. We also plan to extend the tool to provide counter examples when models do not meet a property specification.

Our goal is to develop a tool providing a graphical user interface where libraries of DEBS specifications and property patterns can be used by DEBS developers to validate their systems for use under multiple DEBS middleware and to validate their assumptions with regards to the behaviour of the DEBSs and applications.

References

[1] Property Pattern Mappings for CTL. `http://patterns.projects.cis.ksu.edu/documentation/patterns/ctl.shtml`. SANTOS Laboratory, Foundations, Tools, and Methodologies for High Assurance Software Development, Kansas State University.

[2] C Language Integrated Production System. `http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/areas/expert/systems/clips/0.html`, February 1995. NASA Software Technology Branch, Lyndon B. Johnson Space Center.

[3] Mobility Model Checker. `http://www.cs.sunysb.edu/~lmc/mmc/`, 2003. The Logic-Based Model Checking Project.

[4] W3C XML Schema. `http://www.w3.org/XML/Schema`, 2004. World Wide Web Consortium.

[5] UML 2.1.1 Superstructure Specification, Chapter 15: State Machines. `http://www.omg.org/technology/documents/formal/uml.htm`, February 2007. Object Management Group OMG.

[6] JoCaml Version 3.11. `http://jocaml.inria.fr`, December 2008. Institut National de Recherche en Informatique et Automatique.

[7] Objective Caml Version 3.11. `http://caml.inria.fr`, December 2008. Institut National de Recherche en Informatique et Automatique.

[8] *Proceedings of the 2nd International Conference on Distributed Event-Based Systems (DEBS'08)*, Rome, Italy, July 2008. ACM Press. Chair Roberto Baldoni.

[9] *Proceedings of the 3rd International Conference on Distributed Event-Based Systems (DEBS'09)*, Nashville, TN, US, July 2009. ACM Press. Chairs Douglas C. Schmidt and Aniruddha Gokhale.

[10] Projects Utilizing NaradaBrokering. `http://www.naradabrokering.org/deployments/index.html`, November 2009. Pervasive Technology Labs at Indiana University.

[11] The XSB Logic Programming System, Version 3.2 (Kopi Lewak). `http://xsb.sourceforge.net/`, March 2009. XSB Research Group.

[12] Roberto M. Amadio, Ilaria Castellani, and Davide Sangiorgi. On Bisimulations for the Asynchronous pi-Calculus. In *CONCUR '96: Proceedings of the 7th International Conference on Concurrency Theory*, pages 147–162, London, UK, 1996. Springer-Verlag.

[13] Charles Antony and Richard Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[14] Jean Bacon, Ken Moody, John Bates, Richard Hayton, Chaoying Ma, Andrew McNeil, Oliver Seidel, and Mark Spiteri. Generic Support for Distributed Applications. *Computer*, 33(3):68–76, 2000.

[15] Jos Baeten, Dirk Albert van Beek, and J.E. Rooda. Process Algebra. In Paul A. Fishwick, editor, *Handbook on Dynamic System Modeling*, pages 19.1 – 19.21. Chapman & Hall/CRC, 2007.

[16] Roberto Baldoni, Roberto Beraldi, S. Tucci Piergiovanni, and Antonio Virgillito. On the Modelling of Publish/Subscribe Communication Systems. *Concurrency and Computation: Practice and Experience*, 17(12):1471–1495, 2005.

[17] Franck Barbier and Nicolas Belloir. Component Behavior Prediction and Monitoring Through Built-in Test. In *ECBS '03: Proceedings of the 10th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, pages 17–22, Huntsville, AL, USA, April 2003. IEEE Computer Society.

[18] Jan A. Bergstra and Jan Willem Klop. Algebra of Communicating Processes with Abstraction. *Theoretical Computer Science*, 37:77–121, 1985.

[19] Fredrick B. Beste. The Model Prover — A sequent-calculus based modal mu-calculus model checker tool for finite control pi-calculus agents. Master's thesis, Uppsala University, Sweden, January 1998.

[20] Sumeer Bhola, Robert E. Strom, Saurabh Bagchi, Yuanyuan Zhao, and Joshua S. Auerbach. Exactly-once Delivery in a Content-based Publish-Subscribe System. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 7–16, Washington, DC, USA, 2002. IEEE Computer Society.

[21] Philippe Bidinger, Alan Schmitt, and Jean-Bernard Stefani. An Abstract Machine for the Kell Calculus. In Martin Steffen and Gianluigi Zavattaro, editors, *7th IFIP Internaional Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, volume 3535 of *Lecture Notes in Computer Science*, pages 31–46. Lecture Notes in Computer Science, Springer-Verlag, 2005.

[22] Rolando Blanco and Paulo Alencar. Event Models in Distributed Event Based Systems. In Anika Hinze and Alejandro Buchmann, editors, *Principle and Applications of Distributed Event-based Systems*. IGI Global, 2010. To appear.

[23] Rolando Blanco, Jun Wang, and Paulo Alencar. A Metamodel for Distributed Event Based Systems. In *Proceedings of the 2nd International Conference on Distributed Event-Based Systems (DEBS'08)* [8], pages 221–232. Chair Roberto Baldoni.

[24] Cristiana Bolchini, Carlo Curino, Elisa Quintarelli, Fabio A. Schreiber, and Letizia Tanca. A Data-oriented Survey of Context Models. *SIGMOD Record*, 36(4):19–26, 2007.

[25] Gérard Boudol. Asynchrony and the Pi-calculus. Technical Report RR-1702, INRIA Sophia Antipolis, 1992.

[26] Julian C. Bradfield. Simplifying the Modal Mu-Calculus Alternation Hierarchy. In *STACS '98: Proceedings of the 15th Annual Symposium on Theoretical Aspects of Computer Science*, pages 39–49, London, UK, 1998. Springer-Verlag.

[27] Julian C Bradfield and Colin Stirling. Modal Logics and Mu-Calculi: An Introduction. In Jan A. Bergstra, Alban Ponse, and Scott A. Smolka, editors, *Handbook of Process Algebra*, chapter 1.4, pages 293–330. Elsevier Science, March 2001.

[28] Michele Bugliesi, Giuseppe Castagna, and Silvia Crafa. Boxed Ambients. In *TACS 2001, 4th. International Symposium on Theoretical Aspects of Computer Science*, number 2215 in Lecture Notes in Computer Science, pages 38–63, Sendai, Japan, 2001. Springer-Verlag.

[29] Tevfik Bultan. Fixpoints. `http://www.cs.ucsb.edu/~bultan/courses/267/lectures/l3.ppt`, 2008. CS 267, Graduate Course in Automated Verification, Lecture 3, Department of Computer Science, University of California, Santa Barbara.

[30] Tevfik Bultan, Xiang Fu, Richard Hull, and Jianwen Su. Conversation Specification: A New Approach to Design and Analysis of E-Service Composition. In *WWW '03: Proceedings of the 12th International Conference on World Wide Web*, pages 403–410, New York, NY, USA, 2003. ACM Press.

[31] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *A System of Patterns - Pattern Oriented Software Architecture*. Wiley, 1996.

[32] L. Ruhai Cai, Jeremy S. Bradbury, and Jürgen Dingel. Verifying Distributed, Event-Based Middleware Applications Using Domain-Specific Software Model Checking. In Marcello M. Bonsangue and Einar Broch Johnsen, editors, *FMOODS*, volume 4468 of *Lecture Notes in Computer Science*, pages 44–58. Springer-Verlag, June 2007.

[33] Luís Caires and Luca Cardelli. A Spatial Logic for Concurrency (Part I). *Information and Computation*, 186(2):194–235, 2003.

[34] Luca Cardelli and Andrew D. Gordon. Mobile Ambients. In *FoSSaCS '98: Proceedings of the First International Conference on Foundations of Software Science and Computation Structure*, pages 140–155, London, UK, 1998. Springer-Verlag.

[35] Luca Cardelli and Andrew D. Gordon. Anytime, Anywhere: Modal Logics for Mobile Ambients. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 365–377, New York, NY, USA, 2000. ACM.

[36] Luca Cardelli and Andrew D. Gordon. Logical Properties of Name Restriction. In Samson Abramsky, editor, *TLCA 2001: Proceedings of the 5th International Conference of Typed Lambda Calculi and Applications*, volume 2044 of *Lecture Notes in Computer Science*, pages 46–60. Springer-Verlag, 2001.

[37] Jan Carlson and Björn Lisper. An Event Detection Algebra for Reactive Systems. In *EMSOFT '04: Proceedings of the 4th ACM International Conference on Embedded Software*, pages 147–154, New York, NY, USA, 2004. ACM Press.

[38] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems*, 19(3):332–383, 2001.

[39] K. Mani Chandy and Jayadev Misra. *Parallel program design: a foundation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

[40] Witold Charatonik, Andrew D. Gordon, and Jean-Marc Talbot. Finite-Control Mobile Ambients. In *ESOP '02: Proceedings of the 11th European Symposium on Programming Languages and Systems*, pages 295–313, London, UK, 2002. Springer-Verlag.

[41] Witold Charatonik, Silvano Dal Zilio, Andrew D Gordon, Supratik Mukhopadhyay, and Jean-Marc Talbot. Model Checking Mobile Ambients. *Theoretical Computer Science*, 308(1-3):277–331, 2003.

[42] Weidong Chen and David S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43(1):20–74, 1996.

[43] Mariano Cilia, Michael Haupt, Mira Mezini, and Alejandro Buchmann. The Convergence of AOP and Active Databases: Towards Reactive Middleware. In *GPCE '03: Proceedings of the Second International Conference on Generative Programming and Component Engineering*, pages 169–188, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[44] Edmund M. Clarke and E. Allen Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs, Workshop*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71, London, UK, 1982. Springer-Verlag.

[45] World Wide Web Consortium. Web Services Eventing (WS-Eventing). `http://www.w3.org/Submission/WS-Eventing/`, March 2006. Member Submission.

[46] Gianpaolo Cugola and H.-Arno Jacobsen. Using Publish/Subscribe Middleware for Mobile Systems. *ACM SIGMOBILE Mobile Computing and Communications Review*, 6(4):25–33, 2002.

[47] Gianpaolo Cugola, Elisabetta Di Nitto, and Alfonso Fuggetta. Exploiting an Event-Based Infrastructure to Develop Complex Distributed Systems. In *ICSE '98: Proceedings of the 20th International Conference on Software Engineering*, pages 261–270, Washington, DC, USA, 1998. IEEE Computer Society.

[48] Gianpaolo Cugola, Elisabetta Di Nitto, and Alfonso Fuggetta. The JEDI Event-Based Infrastructure and Its Application to the Development of the OPSS WFMS. *IEEE Transactions On Software Engineering*, 27(9):827–850, September 2001.

[49] Mads Dam. Model Checking Mobile Processes. *Information and Computation*, 129(1):35–51, 1996.

[50] Mads Dam. Proof Systems for Pi-Calculus Logics. In Ruy J.G.B. de Queiroz, editor, *Logic for Concurrency and Synchronisation*, pages 145–212. Kluwer Academic Publishers, 2003. Trends in Logic – Studia Logica Library.

[51] Luca de Alfaro and Thomas A. Henzinger. Interface Automata. *ACM SIGSOFT Software Engineering Notes*, 26(5):109–120, 2001.

[52] François Désarménien. Perl CPAN module Parse::Yapp. `http://search.cpan.org/~fdesar/Parse-Yapp-1.05/lib/Parse/Yapp.pm`, 2001.

[53] Anind K. Dey. Understanding and Using Context. *Personal Ubiquitous Computing*, 5(1):4–7, 2001.

[54] Marcio S. Dias and Marlon E. R. Vieira. Software Architecture Analysis Based on Statechart Semantics. In *IWSSD '00: Proceedings of the 10th International Workshop on Software Specification and Design*, Washington, DC, USA, 2000. IEEE Computer Society.

[55] Jürgen Dingel, David Garlan, Somesh Jha, and David Notkin. Reasoning About Implicit Invocation. In *SIGSOFT '98/FSE-6: Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 209–221, New York, NY, USA, 1998. ACM Press.

[56] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in Property Specifications for Finite-State Verification. In *Proceedings of the 21st International Conference on Software Engineering*, volume 0, pages 411–420, Los Alamitos, CA, USA, May 1999. IEEE Computer Society.

[57] E. Allen Emerson. Model Checking and the Mu-calculus. In Neil Immerman and Phokion G. Kolaitis, editors, *Descriptive Complexity and Finite Models*, volume 31 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 185–214. American Mathematical Society, 1996.

[58] E. Allen Emerson and Edmund M. Clarke. Characterizing Correctness Properties of Parallel Programs Using Fixpoints. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, pages 169–181, London, UK, 1980. Springer-Verlag.

[59] E. Allen Emerson and Joseph Y. Halpern. "Sometimes" and "Not Never" Revisited: On Branching Versus Linear Time. *Journal of the Association for Computer Machinery*, 33(1):151–178, 1986.

[60] Opher Etzion. Semantic Approach to Event Processing. In Jacobsen et al. [89], page 139. Invited talk `http://www.debs.msrg.utoronto.ca/etzion.pdf`.

[61] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

[62] Mohamed E. Fayad, Douglas C. Schmidt, and Ralph E. Johnson. *Implementing Application Frameworks: Object-Oriented Frameworks at Work*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[63] Pascal Fenkam, Harald Gall, and Mehdi Jazayeri. A Systematic Approach to the Development of Event Based Applications. In *22nd Symposium on Reliable Distributed Systems (SRDS 2003)*, page 199. IEEE Computer Society, October 2003.

[64] Pascal Fenkam, Mehdi Jazayeri, and Gerald Reif. On Methodologies for Constructing Correct Event-Based Applications. In Antonio Carzaniga and Pascal Fenkam, editors, *Proceedings of the 3rd International Workshop on Distributed Event-Based Systems (DEBS'04)*, pages 38–43, Edinburgh, Scotland, UK, May 2004. IEEE Computer Society.

[65] Gian-Luigi Ferrari, Stefania Gnesi, Ugo Montanari, and Marco Pistore. A Model-Checking Verification Environment for Mobile Processes. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 12(4):440–473, 2003.

[66] Gian-Luigi Ferrari, Stefania Gnesi, Ugo Montanari, and Marco Pistore. A Model-Checking Verification Environment for Mobile Processes. *ACM Transactions on Software Engineering and Methodology*, 12(4):440–473, 2003.

[67] Ludger Fiege. *Visibility in Event-Based Systems*. PhD thesis, Department of Computer Science, Darmstadt University of Technology, Darmstadt, Germany, Apr 2005.

[68] Ludger Fiege, Mira Mezini, Gero Mühl, and Alejandro P. Buchmann. Engineering Event-Based Systems with Scopes. In Boris Magnusson, editor, *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP02)*, volume 2374 of *Lecture Notes in Computer Science*, pages 309–333. Springer-Verlag, June 2002.

[69] Ludger Fiege, Gero Mühl, and Felix C. Gärtner. Modular Event-Based Systems. *The Knowledge Engineering Review*, 17(4):359–388, 2002.

[70] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A Calculus of Mobile Agents. In *CONCUR '96: Proceedings of the 7th International Conference on Concurrency Theory*, pages 406–421, London, UK, 1996. Springer-Verlag.

[71] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[72] David Garlan, Serge Khersonsky, and Jung Soo Kim. Model Checking Publish-Subscribe Systems. In *Proceedings of The 10th International SPIN Workshop on Model Checking of Software (SPIN 03)*, Portland, Oregon, USA, 2003.

[73] David Garlan and Curtis Scott. Adding Implicit Invocation to Traditional Programming Languages. In *ICSE '93: Proceedings of the 15th International Conference on Software Engineering*, pages 447–455, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[74] David Garlan and Mary Shaw. An Introduction to Software Architecture. Technical Report CMU-CS-94-166, School of Computer Science, Carnegie Mellon University, January 1994.

[75] James Garson. Modal Logic. In Edward N. Zalta, editor, *Stanford Encyclopedia of Philosophy*. The Metaphysics Research Lab, Stanford University, May 2008. Available at `http://plato.stanford.edu/entries/logic-modal/`.

[76] Kurt Geihs. Middleware Challenges Ahead. *Computer*, 34(6):24–31, 2001.

[77] Diogo Guerra, Ute Gawlick, and Pedro Bizarro. An Integrated Data Management Approach to Manage Health Care Sensor Data. In *Proceedings of the 3rd ACM International Conference on Distributed Event-Based Systems (DEBS'09)* [9]. Chairs Douglas C. Schmidt and Aniruddha Gokhale.

[78] David Harel. Statecharts: a Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[79] Matthew Hennessy. *Algebraic Theory of Processes*. MIT Press, Cambridge, MA, USA, 1988.

[80] Matthew Hennessy. *A Distributed Pi-Calculus*. Cambridge University Press, New York, NY, USA, 2007.

[81] Matthew Hennessy and Robin Milner. On Observing Nondeterminism and Concurrency. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, pages 299–309, London, UK, 1980. Springer-Verlag.

[82] Matthew Hennessy and James Riely. Resource Access Control in Systems of Mobile Agents. Technical Report 2/98, Computer Science, School of Cognitive and Computing Sciences, University of Sussex, 1998.

[83] Matthew Hill, Murray Campbell, Yuan-Chi Chang, and Vijay Iyengar. Event Detection in Sensor Networks for Modern Oil Fields. In *Proceedings of the 2nd International Conference on Distributed Event-Based Systems (DEBS'08)* [8], pages 95–102. Chair Roberto Baldoni.

[84] Annika Hinze, Kai Sachs, and Alejandro Buchmann. Event-Based Applications and Enabling Technologies. In *Proceedings of the 3rd ACM International Conference on Distributed Event-Based Systems (DEBS'09)* [9]. Chairs Douglas C. Schmidt and Aniruddha Gokhale.

[85] Daniel Hirschkoff, Étienne Lozes, and Davide Sangiorgi. Separability, Expressiveness, and Decidability in the Ambient Logic. In *LICS '02: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 423–432, Washington, DC, USA, 2002. IEEE Computer Society.

[86] Daniel Hirschkoff, Tom Hirschowitz, Damien Pous, Alan Schmitt, and Jean-Bernard Stefani. Component-Oriented Programming with Sharing: Containment is Not Ownership. In Robert Glück and Michael R. Lowry, editors, *4th International Conference on Generative Programming and Component Engineering GPCE*, Lecture Notes in Computer Science, pages 389–404. Lecture Notes in Computer Science, Springer-Verlag, 2005.

[87] Kohei Honda and Mario Tokoro. An Object Calculus for Asynchronous Communication. In *ECOOP '91: Proceedings of the European Conference on Object-Oriented Programming*, pages 133–147, London, UK, 1991. Springer-Verlag.

[88] Hans-Arno Jacobsen, Alex Cheung, Guoli Li, Balasubramaneyam Maniymaran, Vinod Muthusamy, and Reza Sherafat Kazemzadeh. The PADRES Publish/Subscribe System. In Anika Hinze and Alejandro Buchmann, editors, *Principle and Applications of Distributed Event-based Systems*. IGI Global, 2010. To appear.

[89] Hans-Arno Jacobsen, Gero Mühl, and Michael A. Jaeger, editors. *Proceedings of the Inaugural Conference on Distributed Event-Based Systems*, New York, NY, USA, June 2007. ACM Press.

[90] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag, June 1997.

[91] Prabhudev Konana, Guangtian Liu, Chan-Gun Lee, and Honguk Woo. Specifying Timing Constraints and Composite Events: An Application in the Design of Electronic Brokerages. *IEEE Transactions on Software Engineering*, 30(12):841–858, 2004. Member-Mok, Aloysius K.

[92] Dexter Kozen. Results on the Propositional mu-Calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[93] Balachander Krishnamurthy and David S. Rosenblum. Yeast: A General Purpose Event-Action System. *IEEE Transactions on Software Engineering*, 21(10):845–857, 1995.

[94] Zakir Laliwala, Vikram Sorathia, and Sanjay Chaudhary. Semantic and rule based event-driven services-oriented agricultural recommendation system. In *ICDCSW '06: Proceedings of the 26th IEEE International ConferenceWorkshops on Distributed Computing Systems*, page 24, Washington, DC, USA, 2006. IEEE Computer Society.

[95] Nancy Gail Leveson, Mats Per Erik Heimdahl, Holly Hildreth, and Jon Damon Reese. Requirements Specification for Process-Control Systems. *IEEE Transactions on Software Engineering*, 20(9):684–707, 1994.

[96] Francesca Levi and Davide Sangiorgi. Mobile Safe Ambients. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(1):1–69, 2003.

[97] Christoph Liebig, Mariano Cila, and Alejandro Buchmann. Event Composition in Time-dependent Distributed Systems. In *Proceedings of the 4th IFCIS International Conference on Cooperative Information Systems (CoopIS 99)*, pages 70–78. IEEE Computer Society, 1999.

[98] Huimin Lin. Symbolic Bisimulation and Proof Systems for the π-Calculus. Technical Report 7/94, School of Cognitive and Computer Science, University of Sussex, 1994.

[99] David C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[100] David C. Luckham and James Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, 21(9):717–734, 1995.

[101] Emil Lupu, Naranker Dulay, Alberto Schaeffer Filho, Sye Keoh, Morris Sloman, and Kevin Twidle. AMUSE: Autonomic Management of Ubiquitous E-Health Systems. *Concurrency and Computation: Practice and Experience*, 20(3):277–295, 2008.

[102] Qin Ma and Luc Maranget. Algebraic Pattern Matching in Join Calculus. *Logical Methods in Computer Science*, 4(1:7), 2008. No Publisher. Available at `http://www.lmcs-online.org/ojs/viewarticle.php?id=284`.

[103] Angelika Mader. *Verification of Modal Properties Using Boolean Equation Systems*. Edition versal 8, Bertz Verlag, Berlin, 1997.

[104] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying Distributed Software Architectures. In *Proceedings of the 5th European Software Engineering Conference*, pages 137–153, London, UK, 1995. Springer-Verlag.

[105] Masoud Mansouri-Samani and Morris Sloman. GEM: A Generalised Event Monitoring Language for Distributed Systems. *IEE/IOP/BCS Distributed Systems Engineering Journal*, 4(2):96 – 108, June 1997.

[106] Radu Mardare, Corrado Priami, Paola Quaglia, and Oleksandr Vagin. Model Checking Biological Systems Described Using Ambient Calculus. In Vincent Danos and Vincent Schächter, editors, *Computational Methods in Systems Biology*, volume 3082 of *Lecture Notes in Computer Science*, pages 85–103. Springer, 2004.

[107] Radu Mateescu. Local Model-Checking of Modal Mu-Calculus on Acyclic Labeled Transition Systems. In *TACAS '02: Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 281–295, London, UK, 2002. Springer-Verlag.

[108] Ken McMillan. Getting started with smv. `http://www.kenmcmil.com/tutorial.ps`, 1999. Cadence Berkeley Laboratories.

[109] George H. Mealy. A Method for Synthesizing Sequential Circuits. *Bell System Technical Journal*, 35(5), 1955.

[110] René Meier and Vinny Cahill. STEAM: Event-Based Middleware for Wireless Ad Hoc Networks. In *22nd International Conference on Distributed Computing Systems Workshops*, pages 639–644, Los Alamitos, CA, USA, 2002. IEEE Computer Society.

[111] René Meier and Vinny Cahill. Exploiting proximity in event-based middleware for collaborative mobile applications. In Jean-Bernard Stefani, Isabelle M. Demeure, and Daniel Hagimont, editors, *Proceedings of the 4th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS'03)*, volume 2893 of *Lecture Notes in Computer Science*, pages 285–296. Springer-Verlag, 2003.

[112] René Meier and Vinny Cahill. Taxonomy of Distributed Event-Based Programming Systems. *The Computer Journal*, 48(5):602–626, 2005.

[113] René Meier and Vinny Cahill. On Event-Based Middleware for Location-Aware Mobile Applications. *IEEE Transactions on Software Engineering*, 99(PrePrints), 2009.

[114] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Lecture Notes in Computer Science, Springer-Verlag, 1980.

[115] Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, Computer Laboratory, University of Cambridge, 1999.

[116] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, Part I. Technical Report ECS-LFCS-89-85, Computer Science Department, University of Edinburgh, 1989.

[117] Robin Milner and Davide Sangiorgi. Barbed Bisimulation. In *ICALP '92: Proceedings of the 19th International Colloquium on Automata, Languages and Programming*, pages 685–695, London, UK, 1992. Springer-Verlag.

[118] Gero Mühl. Generic Constraints for Content-Based Publish/Subscribe Systems. In Carlo Batini, Fausto Giunchiglia, Paolo Giorgini, and Massimo Mecella, editors, *Proceedings of the 9th International Conference on Cooperative Information Systems (CoopIS '01)*, volume 2172 of *Lecture Notes in Computer Science*, pages 211–225. Springer-Verlag, September 2001.

[119] Gero Mühl. *Large-Scale Content-Based Publish /Subscribe Systems*. PhD thesis, Technische Universität Darmstadt, Darmstadt, Germany, September 2002.

[120] Gero Mühl and Ludger Fiege. Supporting Covering and Merging in Content-Based Publish/Subscribe systems: Beyond Name/Value Pairs. *IEEE Distributed Systems Online (DSOnline)*, 2(7), 2001.

[121] Gero Mühl, Ludger Fiege, and Alejandro P. Buchmann. Filter Similarities in Content-Based Publish/Subscribe Systems. In *ARCS '02: Proceedings of the International Conference on Architecture of Computing Systems*, pages 224–240, London, UK, 2002. Springer-Verlag.

[122] Gero Mühl, Ludger Fiege, and Peter R. Pietzuch. *Distributed Event-Based Systems*. Springer-Verlag, 2006.

[123] Roger Michael Needham. Names. In Sape Mullender, editor, *Distributed Systems*, pages 89–101. ACM Press, 1989.

[124] Rocco De Nicola and Frits Vaandrager. Action Versus State Based Logics for Transition Systems. In *Proceedings of the LITP Spring School on Theoretical Computer Science on Semantics of Systems og Concurrent Processes*, pages 407–419, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[125] David Notkin, David Garlan, William G. Griswold, and Kevin J. Sullivan. Adding Implicit Invocation to Languages: Three Approaches. In *Proceedings of the First JSSST International Symposium on Object Technologies for Advanced Software*, pages 489–510, London, UK, 1993. Springer-Verlag.

[126] OASIS. www.oasis-open.org/committees/wsrm/, 2004. OASIS Web Services Reliable Messaging (WSRM) TC.

[127] David Ogle, Heather Kreger, Abdi Salahshour, Jason Cornpropst, Eric Labadie, Mandy Chessell, Bill Horn, John Gerken, James Schoech, and Mike Wamboldt. Canonical Situation Data Format: The Common Base Event V1.0.1. http://dev.eclipse.org/viewcvs/indextools.cgi/;checkout;/hyades-home/docs/components/common_base_event/cbe101spec/CommonBaseEvent_SituationData_V1.0.1.pdf, 2004. International Business Machines Corporation.

[128] Brian Oki, Manfred Pfluegl, Alex Siegel, and Dale Skeen. The Information Bus: An Architecture for Extensible Distributed Systems. In *SOSP '93: Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 58–68, New York, NY, USA, 1993. ACM Press.

[129] Catuscia Palamidessi. Comparing the Expressive Power of the Synchronous and the Asynchronous π-calculus. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 256–265, New York, NY, USA, 1997. ACM.

[130] Shrideep Pallickara and Geoffrey Fox. NaradaBrokering: A Distributed Middleware Framework and Architecture for Enabling Durable Peer-to-Peer Grids. In *Proceedings of ACM/IFIP/USENIX 2003 International Middleware Conference Middleware*, volume 2672 of *Lecture Notes in Computer Science*, pages 41–61, Berlin, Germany, 2003. Lecture Notes In Computer Science, Springer-Verlag.

[131] D.M.R. Park. Fixpoint Induction and Proof of Program Semantics. *Machine Intelligence*, 5:59–78, 1969.

[132] Joachim Parrow. An Introduction to the Pi-Calculus. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, pages 479–543. Elsevier, 2001.

[133] Peter Pietzuch, David Eyers, Samuel Kounev, and Brian Shand. Towards a Common API for Publish/Subscribe. In Jacobsen et al. [89], pages 152–157.

[134] Peter Robert Pietzuch. *Hermes: A Scalable Event-Based Middleware*. PhD thesis, University of Cambridge, Queens' College, February 2004.

[135] Amir Pnueli. The Temporal Logic of Programs. In *SFCS '77: Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[136] Paola Quaglia and David Walker. On Synchronous and Asynchronous Mobile Processes. In *FOSSACS '00: Proceedings of the Third International Conference on Foundations of Software Science and Computation Structures*, pages 283–296, London, UK, 2000. Springer-Verlag.

[137] Y.S. Ramakrishna, C.R. Ramakrishnan, I.V. Ramakrishnan, Scott A. Smolka, Terrance Swift, and David S. Warren. Efficient Model Checking Using Tabled Resolution. In *CAV '97: Proceedings of the 9th International Conference on Computer Aided Verification*, pages 143–154, London, UK, 1997. Springer-Verlag.

[138] Rapide Design Team. DRAFT Rapide 1.0 Pattern Language Reference Manual. `http://pavg.stanford.edu/rapide/lrms/patterns.ps`, July 1997. Program Analysis and Verification Group, Stanford University.

[139] Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: An Extensible and Highly-Modular Software Model Checking Framework. In *ESEC/FSE-11: Proceedings of the 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 267–276, New York, NY, USA, 2003. ACM.

[140] David S. Rosenblum and Alexander L. Wolf. A Design Framework for Internet-Scale Event Observation and Notification. In *ESEC '97/FSE-5: Proceedings of the 6th European Conference held jointly with the 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 344–360, New York, NY, USA, 1997. Springer-Verlag New York, Inc.

[141] Szabolcs Rozsnyai, Josef Schiefer, and Alexander Schatten. Concepts and Models for Typing Events for Event-Based Systems. In Jacobsen et al. [89].

[142] Minsoo Ryu, Jimin Kim, and Ji Chan Maeng. Reentrant Statecharts for Concurrent Real-Time Systems. In Hamid R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications & Conference on Real-Time Computing Systems and Applications*, pages 1007–1013. CSREA Press, June 2006.

[143] Konstantinos Sagonas and Terrance Swift. An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. *ACM Transactions on Programming Languages and Systems*, 20(3):586–634, 1998.

[144] Davide Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis, University of Edinburgh, Scotland, UK, 1992.

[145] Davide Sangiorgi. From pi-calculus to Higher-Order pi-calculus - and Back. In *TAPSOFT '93: Proceedings of the International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, pages 151–166, London, UK, 1993. Springer-Verlag.

[146] Davide Sangiorgi. Bisimulation: From the Origins to Today. In Harald Ganzinger, editor, *Proceedings of the Nineteenth Annual IEEE Symp. on Logic in Computer Science, LICS 2004*, pages 298–302. IEEE Computer Society Press, July 2004. Invited Talk.

[147] Ivan Scagnetto and Marino Miculan. Ambient Calculus and its Logic in the Calculus of Inductive Constructions. *Electronic Notes in Theoretical Computer Science*, 70(2), 2002.

[148] Josef Schiefer, Szabolcs Rozsnyai, Christian Rauscher, and Gerd Saurer. Event-Driven Rules for Sensing and Responding to Business Situations. In Jacobsen et al. [89], pages 198–205.

[149] Alan Schmitt and Jean-Bernard Stefani. The M-Calculus: a Higher-Order Distributed Process Calculus. *ACM SIGPLAN Notices*, 38(1):50–61, 2003.

[150] Alan Schmitt and Jean-Bernard Stefani. The Kell Calculus: A Family of Higher-Order Distributed Process Calculi. In Corrado Priami and Paola Quaglia, editors, *Global Computing*, volume 3267 of *Lecture Notes in Computer Science*, pages 146–178. Springer-Verlag, 2004.

[151] Anthony J H Simons. On the Compositional Properties of UML Statechart Diagrams. In *Proceedings of the Third Workshop on Rigorous Object-Oriented Methods, (ROOM2000)*, Electronic Workshops in Computing (eWiC), York, UK, January 2000. The British Computer Society (BCS).

[152] Jatinder Singh and Jean Bacon. Event-Based Data Control In Healthcare. In *Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion*, pages 84–86, New York, NY, USA, 2008. ACM.

[153] Thirunavukkarasu Sivaharan, Gordon Blair, and Geoff Coulson. GREEN: A Configurable and Re-configurable Publish-Subscribe Middleware for Pervasive Computing. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA)*, volume 3760, pages 732–749, Berlin, Germany, 2005. Lecture Notes In Computer Science, Springer-Verlag.

[154] Hosung Song and Kevin J. Compton. Verifying the π-calculus by Promela Translation. Technical Report CSE-TR-472-03, Department of Electrical Engineering and Computer Science, Univeristy of Michigan, Ann Arbor, 2003.

[155] Jean-Bernard Stefani. A Calculus of Kells. In *In Proceedings 2nd International Workshop on Foundations of Global Computing*, volume 85. Electronic Notes in Theoretical Computer Science, Elsevier, 2003.

[156] SUN-JINI. JINI's Distributed Events Specification, Version 1.0. `http://java.sun.com/products/jini/2.1/doc/specs/html/event-spec.html`, 2003. Sun Microsystems, Inc.

[157] SUN-JMS. Java Message Service (JMS) Specification, Version 1.1. `http://java.sun.com/products/jms/docs.html`, 2002. Sun Microsystems, Inc.

[158] Alfred Tarski. A Lattice-Theoretical Fixpoint Theorem and Its Applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[159] Margus Veanes, Colin Campbell, Wolfram Schulte, and Pushmeet Kohli. On-The-Fly Testing of Reactive Systems. Technical Report MSR-TR-2005-05, Microsoft Research, January 2005.

[160] Björn Victor. *A Verification Tool for the Polyadic π-Calculus*. Licentiate thesis, Department of Computer Systems, Uppsala University, Sweden, May 1994. Available as report DoCS 94/50.

[161] Björn Victor and Faron Moller. The Mobility Workbench — A Tool for the pi-Calculus. In David Dill, editor, *CAV '94: Proceedings of the 6th International Conference on Computer Aided Verification*, volume 818, pages 428–440, London, UK, 1994. Springer-Verlag.

[162] Péter Völgyesi, Miklós Maróti, Sebestyen Dóra, Esteban Osses, and Ákos Lédeczi. Software Composition and Verification for Sensor Networks. *Science of Computing Programming*, 56(1-2):191–210, 2005.

[163] Michael von der Beeck. A Comparison of Statecharts Variants. In *ProCoS: Proceedings of the Third International Symposium Organized Jointly with the Working Group Provably Correct Systems on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 128–148, London, UK, 1994. Springer-Verlag.

[164] Michael von der Beeck. A Concise Compositional Statecharts Semantics Definition. In *FORTE/PSTV 2000: Proceedings of the FIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XIII) and Protocol Specification, Testing and Verification (PSTV XX)*, pages 335–350, Deventer, The Netherlands, 2000. Kluwer, B.V.

[165] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl*. O'Reilly, third edition, 2000.

[166] Mark Weiser. Some Computer Science Issues in Ubiquitous Computing. *Communications of the ACM*, 36(7):75–84, 1993.

[167] Jennifer Widom and Stefano Ceri, editors. *Active Database Systems: Triggers and Rules For Advanced Database Processing*. Morgan Kaufmann, 1996.

[168] Jan Wielemaker. An Overview of the SWI-Prolog Programming Environment. In Fred Mesnard and Alexander Serebenik, editors, *Proceedings of the 13th International Workshop on Logic Programming Environments*, pages 1–16, Heverlee, Belgium, december 2003. Katholieke Universiteit Leuven. CW 371.

[169] Roel J. Wieringa. *Design Methods for Software Systems: YOURDON, Statemate and UML*. Morgan Kaufmann Publishers, San Francisco, CA, USA, 2002.

[170] Ping Yang. *Verification Techniques for Mobile Processes and Security Protocols*. PhD thesis, Stony Brook University, Stony Brook, NY, 2006.

[171] Ping Yang, C. R. Ramakrishnan, and Scott A. Smolka. A Logical Encoding of the Pi-Calculus: Model Checking Mobile Processes Using Tabled Resolution. In *VMCAI 2003: Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 116–131, London, UK, 2003. Springer-Verlag.

[172] Hongyu Zhang, Jeremy S. Bradbury, James R. Cordy, and Juergen Dingel. Using Source Transformation to Test and Model Check Implicit-Invocation Systems. *Science of Computer Programming*, 62(3):209–227, 2006.

[173] Yuanyuan Zhao, Daniel Sturman, and Sumeer Bhola. Subscription Propagation in Highly-Available Publish/Subscribe Middleware. In *Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*, pages 274–293, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

APPENDICES

Appendix A

Grammars for High-Level kell-m and kμ

Terminals in the grammars are in boldface and single-quoted. Table A.1 shows special symbols used in the grammars and their kell-m and kμ equivalents.

Tool	**kell-m, kμ**
`->`	$\triangleright$
`->>`	$\diamond$
`~`	$\neg$
`!=`	$\neq$
`&&`	$\wedge$ (kμ)
`||`	$\vee$ (kμ)
`=>`	$\Rightarrow$ (kμ)
`and`	$\wedge$ (kell-m)
`or`	$\vee$ (kell-m)
`=`{`S`}	$\mathcal{K}$
`>=`{`S`}	$\supseteq \mathcal{K}$
`zero`	**0**

Tool	**kell-m, kμ**
`~`{`S`}	$\not\subseteq \mathcal{K}$
`a(x)?`	$a(x)$
`a(x)!`	$\overline{a}(x)$
`a(x)`	$\overleftrightarrow{a}(x)$
`k[x]?`	$k[x]$
`k[x]!`	$\overline{k}[x]$
`k[x]`	$\overleftrightarrow{k}[x]$
`kE`	**E**
`kEe`	$\mathbf{E}_e$
`< action >`	$\langle\alpha\rangle$
`[ action ]`	$[\alpha]$
`A sizeof S`	$\|S\| = A$

Table A.1: Representation of kell-m and kμ Symbols

A.1 High-Level kell-m

What follows is the grammar for the system specification language hl-kell-m introduced in Section 3.3.

Process Definition
process $p(\widetilde{c})$ { Q }

```
1  procdef  ::= 'process' procspec | procspec
2  procspec ::= prochead '{' processk '}'
3  prochead ::= simple_name | simple_name '(' wlst ')'
```

Processes
processk is any process, high-level or kell-m. Rule process_pure is used for kell-m expressions.

```
4  processk         ::= process_pure | process_sugared |
5                       '(' processk ')' | processk '|' processk
6  process_pure     ::= simple_name | abstraction | concretion |
7                       restriction | 'zero'
8  process_sugared  ::= appl_abstr | appl_concr | freshdecl |
9                       vardecl | varassgn | varassgnsync |
10                      moduledef | mod_abstr | mod_concr |
11                      interdef | ifstmt | lstmatch |
12                      lstforeach
```

Unsugared Abstractions and Concretions
`->` is used in triggers for ▷ and `->>` for ◇.

```
13 abstraction         ::= read_trigger | rec_read_trigger |
14                         passiv_trigger | rec_passiv_trigger |
15 concretion          ::= channel_usage | kell_spec
16 kell_spec           ::= name '[' processk ']'
17 channel_usage       ::= name '(' wlst ')'
18 read_trigger        ::= channel_usage '->' processk
19 rec_read_trigger    ::= channel_usage '->>' processk
20 passiv_trigger      ::= kell_spec '->' processk
21 rec_passiv_trigger  ::= kell_spec '->>' processk
```

Restrictions and Fresh Names

```
22 restriction ::= 'new' varlst_nempty process
23 freshdecl   ::= 'fresh' varlst_nempty process
```

Application

$@foo.bar(\widetilde{ws})$, $@foo.bar(\widetilde{ws})(\widetilde{c}) \triangleright P$, and $foo{:}bar \triangleright P$

```
24 appl_concr ::= '@' channel_usage
25 appl_abstr ::= '@' channel_usage '(' varlst ')' '->' processk |
26                '@' channel_usage '(' varlst ')' '->>' processk |
27                '@' channel_usage '->' processk |
28                '@' channel_usage '->>' processk |
29                simple_name ':' simple_name '->' processk |
30                simple_name ':' simple_name '(' wlst ')'  '->'
31                   processk
```

Variable Declaration and Usage

```
33 vardecl        ::= 'var' varspec_lst
34 varspec_lst    ::= varspec
35 varspec        ::= simple_name | simple_name ':=' rassgn
36 varassgn       ::= lassgn ':=' rassgn
37 varassgnsync   ::= syncassgn '->' process
38 syncassgn      ::= lassgn ':=S' rassgn
39 rassgn         ::= appl_concr | 'null' | name | texpr |
40 lassgn         ::= name | modulevar
41 varlst         ::= empty | varlst_nempty
42 varlst_nempty  ::= simple_name | varlst_nempty ',' simple_name
43 varindir       ::= '*' name | '*' '(' modulevar ')' |
44                    '(' varindir ')'
```

Modules

Notice $@(*var).foo(...)$ and $@foo::cast.bar(...)$ are allowed but $@*var.*foo(...)$ is not.

```
45 moduledef    ::= 'module' moduledecl '{' modulebody '}' |
46                  'module' moduledecl 'implements' simple_name
47                     '{' modulebody '}' |
48                  'module' moduledecl 'extends' simple_name
49                     '{' modulebody '}' |
50                  'module' moduledecl 'implements' simple_name
51                     'extends' simple_name '{' modulebody '}' |
52                  'module' moduledecl 'extends' simple_name
53                     'implements' simple_name '{' modulebody '}'
54 moduledecl  ::= simple_name '(' wlst ')' | simple_name
55 modulebody  ::= vardecl ';' moduleprocs | moduleprocs
56 moduleprocs ::= empty | moduleprocs_nempty
57 moduleprocs_nempty ::= read_trigger ';' | moduleprocs_nempty
58                        read_trigger ';'
59 modulevar   ::= casted_name '.' simple_name
```

```
60 mod_abstr    ::= '@' mod_channel_usage '(' varlst ')' '->'
61                    processk |
62                  '@' mod_channel_usage '(' varlst ')' '->>'
63                    processk |
64                  '@' mod_channel_usage '->'
65                    processk |
66                  '@' mod_channel_usage '->>' processk
67 mod_concr    ::= '@' mod_channel_usage
68 mod_channel_usage ::= casted_mod_channel_usage |
69                       uncasted_mod_channel_usage
70 uncasted_mod_channel_usage ::= name '.' simple_name '(' wlst ')'
71 casted_mod_channel_usage   ::= casted_name '.' simple_name '('
72                                 wlst ')'
```

Interfaces

```
73 interdef ::= 'interface' simple_name '{' moduleprocs_nempty '}' |
74              'interface' simple_name  'extends' simple_name '{'
75                 moduleprocs_nempty '}'
```

Conditionals

```
76 ifstmt          ::= 'if' boolexpr 'then' processkelsifs
77                       'else' processk'fi' |
78                     'if' boolexpr 'then' processkelsifs 'fi'
79 elsifs          ::= empty | elsifs_nonempty
80 elsifs_nonempty ::= elsif_cond | elsifs_nonempty elsif_cond
81 elsif_cond      ::= 'elsif' boolexpr 'then' process
```

Boolean Expressions

```
82 boolexpr ::= boolop | '(' boolexpr ')' | 'not' boolexpr
83              boolexpr 'and' boolexpr |
84              boolexpr 'or' boolexpr |
85 boolop   ::= boolval | mod_concr | appl_concr | mathcomp |
86              strcomp | lstcomp
87 boolval  ::= 'true' | 'false'
```

Lists

`-leq` is used to compare if two lists are identical. `-lne` corresponds to the negation of `-leq`. `-has` is used to determine if an expression, typically a name or variable, is in a list. It is below that the grammar for the **match** list operation, introduced in Section 3.3.5, is defined.

List expressions $[a;b;c], [a,b,c]$ are typically passed in channel writes or are assigned to variables; hence, we only support them as concretions (i.e., expression $([a;b;c](l))$ is unsupported).

```
88  lstexpr          ::= '[' lstelems ']'
89  lstelems         ::= empty | lstelems_nempty
90  lstelems_nempty  ::= wlst_param | lstelems ';' wlst_param |
91                       lstelems ',' wlst_param
92  lstcomp          ::= lstexpr_bool
93  lstexpr_bool     ::= lstcompop '-leq' lstcompop |
94                       lstcompop '-lne' lstcompop |
95                       lstexpr '-has' wlst_param
96  lstcompop        ::= lstexpr | name | appl_concr
97  lstmatch         ::= 'match' wlst_param 'with' lstmbody
98  lstmbody         ::= '[' ']' '->' processk 'or'
99                          simple_name '::' simple_name '->'
100                         processk |
101                      simple_name '::' simple_name '->'
102                         processk 'or' '[' ']' '->'
103                         processk |
104                      '[' ']' '->' processk |
105                      simple_name '::' simple_name '->' process
106 lstforeach      ::= 'foreach' simple_name 'in' wlst_param 'do'
107                         processk 'done'
```

Arithmetic Expressions

Overriding of operand precedence rules is not specified below for readability.

```
108 mathexpr       ::= number | mathexpr_arit | '(' mathexpr ')'
109 mathexpr_arit  ::= mathop '+' mathop | mathop '-' mathop |
110                    mathop '*' mathop | mathop '/' mathop |
111                    mathop '%' mathop | '-' mathop
112 mathop         ::= name | appl_concr | mathexpr
113 mathcomp       ::= mathexpr_bool
114 mathexpr_bool  ::= mathcompop '<' mathcompop |
115                    mathcompop '>' mathcompop |
116                    mathcompop '=' mathcompop |
117                    mathcompop '!=' mathcompop |
118                    mathcompop '>=' mathcompop |
119                    mathcompop '<=' mathcompop
120 mathcompop     ::= mathexpr | name | appl_concr | nullc
121 number         ::= int | float
122 float          ::= int '.' int
123 int            ::= digit | int digit
```

```
124 digit          ::=  '0' |  '1' |  '2' |  '3' | '4' |  '5' |
125                     '6' |  '7' | '8' |  '9'
```

String Expressions

'string' below is a sequence of characters surrounded by double quotes. `-lt`, `-gt`, `-eq`, `-ne`, `-ge`, `-le` are operands for string comparisons. Precedence overriding of the string concatenation operand ('.') is not shown for readability.

```
126 strexpr       ::= 'string' | strop '.' strop | '(' strexpr ')'
127 strop         ::= strexpr | name | appl_concr
128 strcomp       ::= strexpr_bool
129 strexpr_bool  ::= strop '-lt' strop | strop '-gt' strop |
130                   strop '-eq' strop | strop '-ne' strop |
131                   strop '-ge' strop | strop '-le' strop
```

Auxiliary Definitions

'name_uc' is a sequence of letters and numbers, starting with an uppercase letter. Similarly defined is 'name_lc', but starting with a lowercase letter.

```
132 wlst          ::= empty | wlst_nempty
133 wlst_nempty   ::= wlst_param | wlst_nempty  ',' wlst_param
134 wlst_param    ::= processk | nullc | varindir | texpr
135 texpr         ::= mathexpr | strexpr | lstexpr
136 simple_name   ::= 'name_uc' | 'name_lc'
137 casted_name   ::= name  '::'  simple_name | '(' casted_name ')'
138 name          ::= simple_name | varindir
139 nullc         ::= 'null'
```

Precedence Rules

```
140 %left  ':' '}'
141 %right '{'
142 %left  ';'
143 %right '(' '['
144 %left  ')' ']'
145 %left  '|'
146 %right 'var'
147 %right 'new' 'fresh' 'match' 'with'
148 %left  '->' '->>'
149 %left  ','
150 %right ':=' ':=S'
151 %left  '::'
152 %right '*'
153 %right '@' '/'
154 %left  '.'
```

```
155 %left 'or'
156 %left 'and' '<' '>' '<=' '>=' '!=' '-lt' '-gt' '-le' '-ge' '-eq'
157
158 %left '+' '-'
159 %left 'times' 'div' '%' 'strcat'
160 %left 'neg'
161 %left 'not'
```

A.2 High-Level kμ

What follows is the grammar for the property specification language hl-kμ introduced in Section 4.3. Some of the rules in the grammar were already defined for hl-kell-m in the previous section and are not shown.

Notice **zero** is used as a property to identify a processes **zero**, **zero** | **zero**, **zero** | **zero** | $\cdots$ | **zero**.

If no kell containment condition is specifed, * (any kell) is assumed. Similarly, in set modalities, individual modalities within the set must be sorrounded by '(' ')'. If no formula is specified after a modality, `tt` is assumed.

Property Definition

```
1 propdef   ::= 'property' prop_spec '{' prop_form '}'
2 prop_spec ::= simple_name '(' pvarlst ')'
```

Property Body

Rule `prop_form` is equivalent to $\mathcal{F}$ in the kμ syntax presented in Section 4.2.

`inert` is equivalent to [-].`ff` and is used to indicate that no more transitions exist in the LTS. `~` is used to negate formulas; `||` is used for $\vee$ and `&&` for $\wedge$; `=>` is used for implication. `kG` is *globally* **G**, `kU` is *until* **U**, `kW` is *weak until*, `F` is *future*, and `E` is eventually. Existential versions of these are also available: `kUe`, `kWe`, `kFe`, `kEe`. `kFalseBefore` and `kBecomesTrueBetween` are patterns presented in [1].

An overriding of the precedence of '.' for formulas $(a = b).\mathcal{F}$ is not shown for readability.

```
3 prop_form      ::= simple_name
4                    'tt' | 'ff' | 'inert'  | '~' prop_form |
5                    prop_comp '.' prop_form |
6                    prop_form '||' prop_form |
7                    prop_form '&&' prop_form |
8                    prop_form '=>' prop_form |
```

```
 9                    prop_mod | prop_mod '.' prop_form |
10                    prop_prefix '(' prop_form_lst ')' |
11                    prop_form 'kU' prop_form |
12                    prop_form 'kW' prop_form |
13                    prop_form 'kUe' prop_form |
14                    prop_form 'kWe' prop_form |
15                    simple_name '(' plst ')' |
16                    '(' prop_form ')'
17 prop_form_lst ::= prop_form |
18                    prop_form_lst ',' prop_form
19 prop_prefix   ::= 'kG' | 'kF' | 'kE' | 'kFe'| 'kEe' |
20                    'kFalseBefore' | 'kBecomesTrueBetween'
```

Conditions

$\mathcal{C}.\mathcal{F}$. $\mathcal{C}$ can be a condition on names, numbers, strings, or list size and element containment for lists.

```
21 prop_comp ::= '(' plst_param prop_op plst_param ')' |
22               '(' prop_comp ')'
23 prop_op   ::= '=' | '!=' | '<' | '>' | '<=' | '>=' |
24               '-lt' | '-gt' | '-le' | '-ge' | '-eq' | '-ne' |
25               'in' | 'sizeof'
```

Diamond and Box Modalities

$\langle\delta\rangle$, $\langle -\delta\rangle$, $\langle S\rangle$, $\langle -S\rangle$, $[\delta]$, $[-\delta]$, $[S]$, $[-S]$.

```
26 prop_mod             ::= '<' prop_transpec '>' |
27                          '[' prop_transpec ']'
28 prop_transpec        ::= '-' |  prop_tran | '-' prop_tran |
29                          '{' prop_transet '}' |
30                          '-' '{' prop_transet '}' |
31                          '(' prop_transpec ')'
32 prop_transet         ::= empty |
33                          prop_transet_nempty
34 prop_transet_nempty ::= '(' prop_tran ')' |
35                          prop_transet_nempty ',' '(' prop_tran ')'
```

Conditions on Transitions

$\delta \equiv (\alpha, \gamma)$ or $\delta \equiv (\alpha_\tau, \gamma_a, \gamma_c)$. `?` after a channel or kell action specifies an abstraction, `!` specifies a concretion, and no decoration specifies a τ transition.

```
36 prop_tran ::= prop_act  |
37               prop_act ',' prop_kcnd |
38               prop_act ',' prop_kcnd ',' prop_kcnd  |
39               prop_act '?' |
```

```
40                prop_act '?' ',' prop_kcnd |
41                prop_act '!' |
42                prop_act '!' ',' prop_kcnd
```

Conditions on Actions

Notice casting of names is supported. This facilitates the specification of properties on actions implemented by modules.

```
43 prop_act      ::= prop_channel '(' plst ')' |
44                   prop_channel '[' simple_name ']'
45 prop_channel ::= simple_name | simple_name '.' simple_name |
46                   simple_name '::' simple_name '.' simple_name
```

Kell Containment Conditions

Possible conditions γ are $*$, $\mathcal{K}$, $\supseteq \mathcal{K}$, $\not\subseteq \mathcal{K}$, and I where I is a variable.

Rule `simple_name` corresponds to I. When specified the variable is instantiated with the set of kells associated to the action. `=`$\{K_1, K_2, ..., K_n\}$ corresponds to $\mathcal{K}$; `>=`$\{K_1, K_2, ..., K_n\}$ corresponds to $\supseteq \mathcal{K}$; and, `~`$\{K_1, K_2, ..., K_n\}$ corresponds to $\not\subseteq \mathcal{K}$.

```
47 prop_kcnd ::= simple_name | '*' | '=' '{' pvarlst '}' |
48                '>=' '{' pvarlst '}' | '~' '{' pvarlst '}
```

Property Verification Requests

```
49 propver      ::= propver_spec | propver_spec ';'
50 propver_spec ::= propver_kind propform 'for' concretion
51                     propexpect
52 propver_kind ::= 'check' |
53                   'check' 'by' 'reduction' |
54                   'check' 'by' 'lts' |
55 propexpect   ::= empty | 'expect' 'yes' | 'expect' 'no'
56 propform     ::= simple_name | simple_name '(' plst ')'
```

Auxiliary Definitions

```
57 plst           ::= empty | plst_nempty
58 plst_nempty    ::= plst_param |   plst_nempty  ',' plst_param
59 plst_param     ::= 'null' | number | 'string' | simple_name
60 pvarlst        ::= empty  | pvarlst_nempty
61 pvarlst_nempty ::= simple_name | pvarlst_nempty  ',' simple_name
```

Precedence Rules

```
62 %left   '||'
63 %left   '&&' '=>'
64 %left   'kU' 'kW' 'kUe' 'kWe'
65 %right 'kG' 'kF' 'kFe'
66 %left   '.'
67 %right '~'
```

Appendix B

Formal Arguments

B.1 $\mathcal{I}[\![\ \{\},\ P\ ]\!]$ is a MMCπ Expression

As in Section 6.3.3, we will abuse the notation and write $\overline{a}(\widetilde{w}, K_s)$ and $a(\widetilde{c}, K_s)$, with K_s kell containment sets. Moreover, we will consider these expressions valid kell-m expressions, even though kell containment sets should be represented using sequences of names.

Recall a MMCπ expression is a π-calculus expression extended with `code(Op, P)`. Both names and variables can be transmitted in communications. We will use induction on the structure of P and assume no name collisions in P (i.e., $\mathcal{F}$ has been invoked). If $P \equiv \mathbf{0}$, by definition of $\mathcal{I}$, $\mathcal{I}[\![\ B, \mathbf{0}\ ]\!] \stackrel{def}{=} \mathbf{0}$, and $\mathbf{0}$ is trivially a MMCπ process. Similarly, if P is a process variable x, by definition of $\mathcal{I}$, $\mathcal{I}[\![\ B, x\ ]\!] \stackrel{def}{=} \mathbf{0}$, and $\mathbf{0}$ is trivially a MMCπprocess. The other possible cases for P are:

- **new** $a\ Q$, by definition $\mathcal{I}[\![\ B, \mathbf{new}\ a\ Q\ ]\!] \stackrel{def}{=} \mathbf{new}\ a\ \mathcal{I}[\![\ B \cup \{a\}, Q\ ]\!]$. By induction $\mathcal{I}[\![\ B, Q\ ]\!]$ is MMCπ. $\mathcal{I}[\![\ B \cup \{a\}, Q\ ]\!]$ is also a MMCπ because, $\mathcal{I}$ does not impose conditions on B, and adding a name to the set of restricted names B, only affects the tuple of names input and output during channel actions (see the $\mathcal{I}$ definitions for $\overline{a}(\widetilde{w})$ and $a(\widetilde{c}) \triangleright R$ above). Finally, since restriction of a MMCπ is a MMCπ, then $\mathcal{I}[\![\ B,\ P\ ]\!]$ is a MMCπ.

- $\overline{a}(\widetilde{w})$, by definition $\mathcal{I}[\![\ B, \overline{a}(\widetilde{w})\ ]\!] \stackrel{def}{=} \mathit{Hwrite}(\overline{a}(\widetilde{w}, \emptyset), \mathbf{0}, B')$, with $B' = B \cup \{a\}$. Since $\mathit{Hwrite}(\overline{a}(\widetilde{w}, \emptyset), \mathbf{0}, B') \stackrel{def}{=} \mathit{Hconv}(\overline{a}(\widetilde{w}, K_a, \emptyset, B').\mathbf{0}, |\widetilde{w}|, 1)$ with K_a a variable, we need to show $\mathit{Hconv}(\overline{a}(\widetilde{w}, K_a, \emptyset, B').\mathbf{0}, |\widetilde{w}|, 1)$ is a MMCπ. By induction on the length of $\widetilde{w}$:

  - $|\widetilde{w}| = 0$, by Hconv's definition, $\mathit{Hconv}(\overline{a}(K_a, \emptyset, B').\mathbf{0}, 0, 1) = \overline{a}(K_a, \emptyset, B').\mathbf{0}$.

Since K_a is a variable, $\emptyset$ is modelled as an empty list (cf. Section 6.3.3), and there are only names in B', $\overline{a}(K_a, \emptyset, B').\mathbf{0}$ is a MMCπ.

– We now assume $\mathit{Hconv}(\overline{a}(w_1, w_2, ..., w_n, K_a, \emptyset, B').\mathbf{0}, n, 1)$ is a MMCπ. Such a process looks as follows:

$$
\begin{array}{l}
\textbf{new } h_j \ (\\
\quad \textbf{new } h_{j-1} \ (\\
\qquad \cdots (\textbf{new } h_1 \ (\overline{a}(\widetilde{y}, K_a, \emptyset, B').\mathbf{0})) \cdots \\
\quad) \\
)
\end{array}
$$

Assuming in $(w_1, ..., w_n)$ there are j higher order expressions, and being $\widetilde{y}$ the resulting output names after higher order expressions have been replaced. We have $|\widetilde{y}| = n$, and

$$
y_i = \begin{cases} w_i & \text{if } w_i \text{ is not higher-order} \\ h_m & \text{with } m \leq j \text{ and } (h_m, w_i) \in H \\ & \text{and } \nexists l : l \leq n, l \neq i, w_l = h_m, \text{ otherwise} \end{cases}
$$

By induction, such a process is a MMCπ. We need to show for a new w_{n+1}, $\mathit{Hconv}(\overline{a}(w_1, w_2, ..., w_n, w_{n+1}, K_a, \emptyset, B').\mathbf{0}, n+1, 1)$ is a MMCπ as well. If w_{n+1} is a name (i.e., not a higher-order expression), then by the definition of *Hconv*, the resulting process is (changes with respect to the process for n are double-underlined):

$$
\begin{array}{l}
\textbf{new } h_j \ (\\
\quad \textbf{new } h_{j-1} \ (\\
\qquad \cdots (\textbf{new } h_1 \ (\overline{a}(\underline{\underline{\widetilde{y'}}}, k_a, \emptyset, B').\mathbf{0})) \cdots \\
\quad) \\
)
\end{array}
$$

Notice the only difference between the resulting process, and the process for $\mathit{Hconv}(\overline{a}(w_1, w_2, ..., w_n, K_a, \emptyset, B').\mathbf{0}, n, 1)$, is the use of $\widetilde{y'}$ instead of $\widetilde{y}$, where $\widetilde{y'} = y_1, y_2, ..., y_n, w_{n+1}$. Since this change only increases the number of names written by one, the resulting process is a MMCπ. If the new w_{n+1} is a higher-order expression, by the definition of *Hconv*, the resulting process is (changes

are double-underlined):

$$\begin{array}{l}\underline{\underline{\textbf{new } h \ (}}\\ \quad \textbf{new } h_j \ (\\ \quad\quad \textbf{new } h_{j-1} \ (\\ \quad\quad\quad \cdots (\textbf{new } h_1 \ (\overline{a}(\underline{\underline{\widetilde{y''}}}, K_a, \emptyset, B').\mathbf{0})) \cdots \\ \quad\quad)\\ \quad)\\ \underline{\underline{)}}\end{array}$$

Where $\widetilde{y''} = y_1, y_2, ..., y_n, h$, with h a fresh name used as a higher-order indicator and $H = H \cup \{(h, w_{n+1})\}$. Since the changes only introduce a restriction **new** h and a name h to the output list, the resulting expression is a MMCπ expression, and therefore, the following is a MMCπ:

$$Hconv(\overline{a}(w_1, w_2, ..., w_{n+1}, K_a, \emptyset, B').\mathbf{0}, n+1, 1)$$

We have shown $Hconv(\overline{a}(\widetilde{w}, K_a, \emptyset, B').\mathbf{0}, |\widetilde{w}|, 1)$, for $|\widetilde{w}| = 0$ and $|\widetilde{w}| = n+1$ are both a MMCπ; then by induction on the length of $\widetilde{w}$, $Hconv(\ \overline{a}(\widetilde{w}, K_a, \emptyset, B').\mathbf{0}$, $|\widetilde{w}|, 1)$ is a MMCπ for all $|\overline{a}(\widetilde{w})| = n$. And because $Hconv(\overline{a}(\widetilde{w}, K_a, \emptyset, B').\mathbf{0}, |\widetilde{w}|, 1)$ is a MMCπ, then $\mathcal{I}[\![\ B, \overline{a}(\widetilde{w})\ ]\!]$ is a MMCπ, therefore $\mathcal{I}[\![\ B, P\ ]\!]$ is a MMCπ. In general, $Hconv(\overline{a}(\widetilde{w}, K_a, \emptyset, B').R, |\widetilde{w}|, 1)$ is a MMCπ if R is a MMCπ. Therefore, $Hwrite(\overline{a}(\widetilde{w}, \emptyset), R, B\ \{a\})$ is a MMCπ when R is a MMCπ. We will make use of this observation when showing that $\mathcal{I}[\![\ B, K[Q]\ ]\!]$ is a MMCπ.

- $\overline{a}(\widetilde{w}, K_s)$ with K_s a kell containment set. The argument is similar to that of $\overline{a}(\widetilde{w})$, but with K_s instead of $\emptyset$. When $Hconv(\overline{a}(\widetilde{w}, K_a, K_s, B').\mathbf{0}, |\widetilde{w}|, 1)$ with K_a a variable, the resulting expression is MMCπ, because K_s is represented as a list. The only difference is K_s may not be empty, but in such a case K_s as part of a communication is a valid MMCπ.

- $a(\widetilde{c})\triangleright Q$, by definition $\mathcal{I}[\![\ B, a(\widetilde{c})\triangleright Q\ ]\!] \stackrel{def}{=} a(\widetilde{c}, \emptyset, K_c, bnd).\texttt{code}(\texttt{inst}(Q, \widetilde{c}, bnd, Q_\pi), Q_\pi)$ with K_c a variable. Because $\widetilde{c}$ is a sequence, possibly empty, of names and variables, $\emptyset$ is a valid value in MMCπ communications, and bnd is a variable, an expression $a(\widetilde{c}, \emptyset, K_c, bnd).P$ is a MMCπ, if P is a MMCπ.

  `code(Op, R)` is a MMC extension of the π-calculus. It is defined as performing the predicate `Op`, and then behaving as `R`. In our case, `Op` is $\texttt{inst}(Q, \widetilde{c}, bnd, Q_\pi)$, and `R` is Q_π. We need to show that after evaluating $\texttt{inst}(Q, \widetilde{c}, bnd, Q_\pi)$, Q_π is a MMCπ. By definition, $\texttt{inst}(Q, \widetilde{c}, bnd, Q_\pi)$ `:-` $Q_\pi = \mathcal{I}[\![\ B \cup bnd, \mathcal{F}[\![\ \mathcal{H}[\![\ Q\ ]\!]\ ]\!]\ ]\!]$. Recall $\mathcal{H}[\![\ Q\ ]\!]$ replaces higher-order indicators found in Q with their corresponding associated process expressions in H. The result of the instantiation is a kell-m process, which is

then processed by $\mathcal{F}$, so that fresh names are used. The result of the fresh-names processing is again a kell-m process.

We will show $\mathcal{H}[\![\ Q\ ]\!]$ is a kell-m process for any kell-m process Q. Similarly, $\mathcal{F}[\![\ Q\ ]\!]$ can be shown to be a kell-m process (we omit the proof). Based on Q and $\mathcal{H}$'s definition, the possible transformations made to Q are:

- $Q = \mathbf{0}$, by definition $\mathcal{H}[\![\ \mathbf{0}\ ]\!] \stackrel{def}{=} \mathbf{0}$, and $\mathbf{0}$ is trivially a kell-m process.
- $Q = x$, if x is a higher-order indicator, by definition $\mathcal{H}[\![\ x\ ]\!] \stackrel{def}{=} P'$, where P' is the kell-m process associated to x in H. Since P' is a kell-m process then $\mathcal{H}[\![\ h\ ]\!]$ is a kell-m process. If x is not a higher-order indicator, then x is a process variable, and process variables are kell-m processes.
- $Q = d(\widetilde{b}) \triangleright R$, by definition $\mathcal{H}[\![\ d(\widetilde{b})\ ]\!] \triangleright R) \stackrel{def}{=} d(\widetilde{b}) \triangleright \mathcal{H}[\![\ R\ ]\!]$. By structural induction, assuming $\mathcal{H}[\![\ R\ ]\!]$ is a kell-m process, then $d(\widetilde{b}) \triangleright \mathcal{H}[\![\ R\ ]\!]$ is also a kell-m process.
- $Q = K[x] \triangleright R$, by definition $\mathcal{H}[\![\ K[x] \triangleright R\ ]\!] \stackrel{def}{=} K[x] \triangleright \mathcal{H}[\![\ R\ ]\!]$. Assuming, by structural induction, if $\mathcal{H}[\![\ R\ ]\!]$ is a kell-m process, then $K[x] \triangleright \mathcal{H}[\![\ R\ ]\!]$ is also a kell-m process.
- $Q = K[R]$, by definition $\mathcal{H}[\![\ K[R]\ ]\!] \stackrel{def}{=} K[\mathcal{H}[\![\ R\ ]\!]]$. Assuming, by structural induction, that $\mathcal{H}[\![\ R\ ]\!]$ is a kell-m process, then $K[\mathcal{H}[\![\ R\ ]\!]]$ is also a kell-m process.
- $Q = \overline{d}(\widetilde{b})$, trivially a kell-m process, since by definition there is no modification of the kell-m process $\overline{d}(\widetilde{b})$: $\mathcal{H}[\![\ \overline{d}(\widetilde{b})\ ]\!] \stackrel{def}{=} \overline{d}(\widetilde{b})$.
- $Q = R(\widetilde{v})$, by definition $\mathcal{H}[\![\ R(\widetilde{v})\ ]\!] \stackrel{def}{=} \mathcal{H}[\![\ R_d\{\widetilde{v}/\widetilde{y}\}\ ]\!]$ when $R(\widetilde{y}) = R_d$. Since R_d is a kell-m process, by structural induction $\mathcal{H}[\![\ R_d\{\widetilde{v}/\widetilde{y}\}\ ]\!]$ is also a kell-m process.
- $Q = R \mid T$, by definition $\mathcal{H}[\![\ R|T\ ]\!] \stackrel{def}{=} \mathcal{H}[\![\ R\ ]\!] \mid \mathcal{H}[\![\ T\ ]\!]$. By structural induction, assuming $\mathcal{H}[\![\ R\ ]\!]$ and $\mathcal{H}[\![\ T\ ]\!]$ are kell-m processes, then their parallel composition is also a kell-m process.

Because $\mathcal{F}[\![\ \mathcal{H}[\![\ Q\ ]\!]\ ]\!]$ is a kell-m process, by structural induction, $\mathcal{I}[\![\ B \cup bnd, \mathcal{F}[\![\ \mathcal{H}[\![\ Q\ ]\!]\ ]\!]\ ]\!]$ is a MMCπ if:

- $a(\widetilde{c}, K_s) \triangleright Q$, with K_s a kell containment set, possibly empty. The argument is the same as for $a(\widetilde{c}) \triangleright Q$ above. Notice $a(\widetilde{c}, K_s, K_c, bnd).P$ is a MMCπ if P is a MMCπ.
- $K[Q]$, by $\mathcal{I}$'s definition $\mathcal{I}[\![\ B, K[Q]\ ]\!] \stackrel{def}{=} \mathcal{S}[\![\ \mathbf{0}, B, K[Q], \emptyset\ ]\!] + \mathcal{I}[\![\ B, \mathcal{A}[\![\ K[Q]\ ]\!]\ ]\!]$ if Q is not the parallel composition of processes R and T; otherwise,

$$\mathcal{I}[\![\ B, K[Q]\ ]\!] \stackrel{def}{=} \mathcal{S}[\![\ \mathbf{0}, B, K[Q], \emptyset\ ]\!] + \mathcal{I}[\![\ B, \mathcal{A}[\![\ K[R|T]\ ]\!]\ ]\!] + \mathcal{I}[\![\ B, \mathcal{A}[\![\ K[T|R]\ ]\!]\ ]\!]$$

First we will show when $Q = \mathbf{0}$, then $\mathcal{I}[\![\ B, K[Q]\ ]\!]$ is a MMCπ. By definition, $\mathcal{I}[\![\ B, K[\mathbf{0}]\ ]\!] \stackrel{def}{=} \mathcal{S}[\![\ \mathbf{0}, B, K[\mathbf{0}], \emptyset\ ]\!] + \mathcal{I}[\![\ B, \mathcal{A}[\![\ K[\mathbf{0}]\ ]\!]\ ]\!]$. By $\mathcal{S}$'s definition:

$$\begin{array}{rcl} \mathcal{S}[\![\ \mathbf{0}, B, K[\mathbf{0}], \emptyset\ ]\!] & \stackrel{def}{=} & \mathit{Hwrite}(\overline{K}(\mathbf{0}, \emptyset), \mathcal{I}[\![\ B, \mathbf{0}\{\mathbf{0}/K[\mathbf{0}]\}\ ]\!], B \setminus \{K\}) \\ & & \quad + \ \mathcal{S}[\![\ \mathrm{K[0]}, \ B, \ \mathbf{0}, \ \{K\}\ ]\!] \\ & \equiv & \mathit{Hwrite}(\overline{K}(\mathbf{0}, \emptyset), \mathcal{I}[\![\ B, \mathbf{0}\ ]\!], B \setminus \{K\}) \ + \ \mathbf{0} \\ & \equiv & \mathit{Hwrite}(\overline{K}(\mathbf{0}, \emptyset), \mathbf{0}, B \setminus \{K\}) \ + \ \mathbf{0} \end{array}$$

By $\mathcal{A}$'s definition, $\mathcal{I}[\![\ B, \mathcal{A}[\![\ K[\mathbf{0}]\ ]\!]\ ]\!] \equiv \mathcal{I}[\![\ B, \mathbf{0}\ ]\!] \equiv \mathbf{0}$. Therefore:

$$\mathcal{I}[\![\ B, K[\mathbf{0}]\ ]\!] \equiv \mathit{Hwrite}(\overline{K}(\mathbf{0}, \emptyset), \mathbf{0}, B \setminus \{K\}) \ + \ \mathbf{0} \ + \ \mathbf{0}$$

We have already shown $\mathit{Hwrite}(\overline{K}(\mathbf{0}, \emptyset), \mathbf{0}, B \setminus \{K\})$ is a MMCπ. Finally, the choice of $\mathit{Hwrite}(\overline{K}(\mathbf{0}, \emptyset), \mathbf{0}, B \setminus \{K\})$, $\mathbf{0}$, and $\mathbf{0}$ is trivially a MMCπ.

For the general case, we need to show $\mathcal{S}[\![\ \mathbf{0}, B, K[Q], \emptyset\ ]\!]$ is a MMCπ, then we will show $\mathcal{A}[\![\ K[Q]\ ]\!]$ produces a kell-m process. We will argue, by structural induction, that $\mathcal{I}[\![\ B, \mathcal{A}[\![\ K[Q]\ ]\!]\ ]\!]$ is therefore a MMCπ.

By $\mathcal{S}$'s definition, $\mathcal{S}[\![\ \mathbf{0}, B, K[Q], \emptyset\ ]\!] \stackrel{def}{=} \mathit{Hwrite}(\overline{K}(Q, \emptyset), \mathcal{I}[\![\ B, \mathbf{0}\{\mathbf{0}/K[Q]\}\ ]\!], B \setminus \{K\})$ $+ \ \mathcal{S}[\![\ \mathrm{K[Q]}, \ B, \ Q, \ \{K\}\ ]\!]$. Since $\mathcal{I}[\![\ B, \mathbf{0}\{\mathbf{0}/K[Q]\}\ ]\!] \equiv \mathcal{I}[\![\ B, \mathbf{0}\ ]\!] \stackrel{def}{=} \mathbf{0}$, then $\mathcal{S}[\![\ \mathbf{0}, B, K[Q], \emptyset\ ]\!] \equiv \mathit{Hwrite}(\overline{K}(Q, \emptyset), \mathbf{0}, B \setminus \{K\}) + \mathcal{S}[\![\ \mathrm{K[Q]}, \ B, \ Q, \ \{K\}\ ]\!]$. Because $\mathit{Hwrite}(\overline{K}(Q, \emptyset), \mathbf{0}, B \setminus \{K\})$ is a MMCπ, we just need to show:

$$\mathcal{S}[\![\ \mathrm{K[Q]}, \ B, \ Q, \ \{K\}\ ]\!] \text{ is a MMC}\pi.$$

The possible cases for Q are:

- Q is not a kell $G[R]$ or a parallel composition of two processes $R|T$, then by $\mathcal{S}$'s definition, $\mathcal{S}[\![\ K[Q], B, Q, \{K\}\ ]\!] \stackrel{def}{=} \mathbf{0}$, and $\mathbf{0}$ is trivially a MMCπ.
- $G[R]$, by $\mathcal{S}$'s definition, $\mathcal{S}[\![\ K[Q], B, Q, \{K\}\ ]\!] \stackrel{def}{=} \mathit{Hwrite}(\overline{G}(R, \{K\}), \mathcal{I}[\![\ B, K[\mathbf{0}]\ ]\!],$ $B \setminus \{G\}) \ + \ \mathcal{S}[\![\ K[Q], B, R, \{K, G\}\ ]\!]$. We have already shown $\mathcal{I}[\![\ B, K[\mathbf{0}]\ ]\!]$ is a MMCπ therefore, $\mathit{Hwrite}(\overline{G}(R, \{K\}), \mathcal{I}[\![\ B, K[\mathbf{0}]\ ]\!], B \setminus \{G\})$ is a MMCπ (recall, $\mathit{Hwrite}(\overline{a}(\widetilde{w}, K_s), T, B)$ is a MMCπ if T is a MMCπ). Still to show is that $\mathcal{S}[\![\ K[Q], B, R, \{K, G\}\ ]\!]$ is a MMCπ. Interesting cases occur when R is the nesting of multiple kells, possibly composed in parallel. For example $K_1[K_2[...K_n[T]...] \mid K_u[W]]$, where T and W are not parallel composition of processes or another kell. In such cases, the result of $\mathcal{S}$ looks as:

$$\begin{array}{ll} & \mathbf{new}\ h_u\ \overline{K_u}(h_u, \{K, G, K_1\}).\mathcal{I}[\![\ B, K[G[K_1[K_2[...K_n[T]...] \mid \mathbf{0}]]]\ ]\!] \\ + & \mathbf{new}\ h_1\ \overline{K_1}(h_1, \{K, G\}).\mathcal{I}[\![\ B, K[G[\mathbf{0}]]\ ]\!] \\ + & \mathbf{new}\ h_2\ \overline{K_2}(h_2, \{K, G, K_1\}).\mathcal{I}[\![\ B, K[G[K_1[\mathbf{0} \mid K_u[W]]]]\ ]\!] \\ + & \cdots \\ + & \mathbf{new}\ h_n\ \overline{K_n}(h_n, \{K, G, K_1, ..., K_{n-1}\}). \\ & \mathcal{I}[\![\ B, K[G[K_1[K_2[...\mathbf{0}...] \mid K_u[W]]]]\ ]\!] \end{array}$$

With

$$H = H \cup \{ \quad (h_u, W), \\ (h_1, K_2[K_3[...K_n[T]...]] \mid K_u[W]), \\ (h_2, K_3[K_4[...K_n[T]...]] \mid K_u[W]), \\ ..., \\ (h_n, T)\}$$

Names hs are higher-order indicators. The channel outputs $\overline{K_i}(h_i, K_s)$ are MMCπ expressions. For the expressions,

$$\mathcal{I}[\![\ B, K[G[K_1[K_2[...]]]]\ ]\!]$$

$\mathcal{I}$'s definition uses the advance function $\mathcal{A}$:

$$\mathcal{I}[\![\ B, K[G[K_1[K_2[...]]]]\ ]\!] \stackrel{def}{=} \mathcal{I}[\![\ B, \mathcal{A}[\![\ K[G[K_1[K_2[...]]]]\ ]\!]\ ]\!]$$

Assuming $\mathcal{A}$ produces a kell-m process (we will show this later), by structural induction, the encoding $\mathcal{I}$ of such process is a MMCπ. Therefore, $\mathcal{S}[\![\ K[G[R]],\ B,\ G[R],\ \{K\}\ ]\!]$, is the composition, via the π-calculus choice operator $+$, of MMCπs. The choice of MMCπs is trivially a MMCπ.

- $R|T$, by $\mathcal{S}$'s definition, $\mathcal{S}[\![\ K[Q], B, R \mid T\ ]\!] \stackrel{def}{=} \mathcal{S}[\![\ K[Q], B, R, \{K\}\ ]\!] + \mathcal{S}[\![\ K[Q], B, T, \{K\}\ ]\!]$. By structural induction on $\mathcal{S}$, both $\mathcal{S}[\![\ K[Q], B, R, \{K\}\ ]\!]$ and $\mathcal{S}[\![\ K[Q], B, T, \{K\}\ ]\!]$ are a MMCπ, and the choice of two MMCπs is trivially a MMCπ.

We will now show the result of $\mathcal{A}[\![\ K[Q]\ ]\!]$ is a kell-m process. By structural induction on $\mathcal{A}$, the cases for Q are:

- $\mathbf{0}$, by $\mathcal{A}$'s definition $\mathcal{A}[\![\ K[\mathbf{0}]\ ]\!] \stackrel{def}{=} \mathbf{0}$, and $\mathbf{0}$ is a kell-m process.
- **new** $c\ R$, by $\mathcal{A}$'s definition $\mathcal{A}[\![\ K[\mathbf{new}\ c\ R]\ ]\!] \stackrel{def}{=} \mathbf{new}\ c\ \mathcal{A}[\![\ K[R]\ ]\!]$. By structural induction, $\mathcal{A}[\![\ K[R]\ ]\!]$ is a kell-m process, and the restriction of a kell-m process is a kell-m process.
- $\overline{a}(\widetilde{w})$ by $\mathcal{A}$'s definition $\mathcal{A}[\![\ K[\overline{a}(\widetilde{w})]\ ]\!] \stackrel{def}{=} \overline{a}(\widetilde{w}, \{K\})$, and $\overline{a}(\widetilde{w}, \{K\})$ is a kell-m process.
- $\overline{a}(\widetilde{w}, K_s)$ by $\mathcal{A}$'s definition $\mathcal{A}[\![\ K[\overline{a}(\widetilde{w}, K_s)]\ ]\!] \stackrel{def}{=} \overline{a}(\widetilde{w}, K'_s)$, with $K' = K_s \cup \{K\}$, and $\overline{a}(\widetilde{w}, K'_s)$ is a kell-m process.
- $a(\widetilde{c}) \triangleright R$, by $\mathcal{A}$'s definition $\mathcal{A}[\![\ K[a(\widetilde{c}) \triangleright R]\ ]\!] \stackrel{def}{=} a(\widetilde{c}, \{K\}) \triangleright K[R]$. This is a kell-m channel trigger expression.
- $a(\widetilde{c}, K_s) \triangleright R$, by $\mathcal{A}$'s definition $\mathcal{A}[\![\ K[a(\widetilde{c}, K_s) \triangleright R]\ ]\!] \stackrel{def}{=} a(\widetilde{c}, K'_s) \triangleright K[R]$ with $K'_s = K_s \cup \{K\}$. This is a kell-m channel trigger expression.

– $L[x] \triangleright R$, by $\mathcal{A}$'s definition $\mathcal{A}[\![\ K[L[x] \triangleright R]\ ]\!] \stackrel{def}{=} L(x, \{K\}) \triangleright K[R]$. This is a kell-m channel trigger expression.

– $L[R]$, by $\mathcal{A}$'s definition $\mathcal{A}[\![\ K[L[R]]\ ]\!] \stackrel{def}{=} \mathcal{A}[\![\ K[\mathcal{A}[\![\ L[R]\ ]\!]]\ ]\!]$. By structural induction, $\mathcal{A}[\![\ L[R]\ ]\!]$ is a kell-m process. $\mathcal{A}$ eliminates kells with null processes, and lifts name restrictions (**new** c), channel concretions ($\overline{a}(\widetilde{a})$) and abstractions ($a(\widetilde{c}) \triangleright U$, $K[x] \triangleright U$) from nested kells outwards. The invocation $\mathcal{A}[\![\ K[\mathcal{A}[\![\ L[R]\ ]\!]]\ ]\!]$ just lifts to the outside of K whatever expression was, in turn, lifted from $L[R]$. The expression being lifted has the form $\mathbf{0}$, $\mathcal{A}[\![\ K[\mathbf{0}]\ ]\!]$, **new** $c\ R$, $a(\widetilde{c}) \triangleright R$, or $L[x] \triangleright R$. We have already shown that the result of advancing on such expressions is also a kell-m expression.

– $R(\widetilde{w})$, by $\mathcal{A}$'s definition $\mathcal{A}[\![\ K[R(\widetilde{w})]\ ]\!] \stackrel{def}{=} \mathcal{A}[\![\ K[R_d\{\widetilde{w}/\widetilde{y}\}]\ ]\!]$ where $R(\widetilde{y}) \stackrel{def}{=} R_d$. By structural induction $\mathcal{A}[\![\ K[R_d\{\widetilde{w}/\widetilde{y}\}]\ ]\!]$ is a kell-m process.

– $R|T$, R can be one of:

  * $\mathbf{0}$, by $\mathcal{A}$'s definition $\mathcal{A}[\![\ K[\mathbf{0} \mid T]\ ]\!] \stackrel{def}{=} \mathcal{A}[\![\ K[T]\ ]\!]$, and by structural induction, $\mathcal{A}[\![\ kkKT\ ]\!]$ is a kell-m process.
  * **new** $c\ U$, by $\mathcal{A}$'s definition $\mathcal{A}[\![\ K[\mathbf{new}\ c\ U|T]\ ]\!] \stackrel{def}{=} \mathbf{new}\ c\ K[U|T]$. A restriction of a kell-m process is a kell-m process.
  * $\overline{a}(\widetilde{w})$, by $\mathcal{A}$'s definition $\mathcal{A}[\![\ K[\overline{a}(\widetilde{w}) \mid T]\ ]\!] \stackrel{def}{=} \overline{a}(\widetilde{w}, \{K\}) \mid K[T]$. Parallel composition of kell-m processes is a kell-m process.
  * $\overline{a}(\widetilde{w}, K_s)$, by $\mathcal{A}$'s definition $\mathcal{A}[\![\ K[\overline{a}(\widetilde{w}, K_s) \mid T]\ ]\!] \stackrel{def}{=} \overline{a}(\widetilde{w}, K_s \cup \{K\}) \mid K[T]$. Parallel composition of kell-m processes is a kell-m process.
  * $a(\widetilde{c}) \triangleright U$, by $\mathcal{A}$'s definition, $\mathcal{A}[\![\ K[(a(\widetilde{c}) \triangleright U) \mid T]\ ]\!] \stackrel{def}{=} a(\widetilde{c}, \{K\}) \triangleright K[U \mid T]$. Trigger expressions that match channel outputs are kell-m processes.
  * $a(\widetilde{c}, K_s) \triangleright U$, by $\mathcal{A}$'s definition, $\mathcal{A}[\![\ K[(a(\widetilde{c}, K_s) \triangleright U) \mid T]\ ]\!] \stackrel{def}{=} a(\widetilde{c}, K_s \cup \{K\}) \triangleright K[U \mid T]$. Trigger expressions that match channel outputs are kell-m processes.
  * $L[x] \triangleright U$, by $\mathcal{A}$'s definition $\mathcal{A}[\![\ K[(L[x] \triangleright U) \mid T]\ ]\!] \stackrel{def}{=} L(x, \{K\}) \triangleright K[U \mid T]$. Trigger expressions that match kells are kell-m processes.
  * $H[U]$, by $\mathcal{A}$'s definition $\mathcal{A}[\![\ K[H[U] \mid T]\ ]\!] \stackrel{def}{=} \mathcal{A}[\![\ K[\mathcal{A}[\![\ H[U]\ ]\!] \mid T]\ ]\!]$. By structural induction $\mathcal{A}[\![\ H[U]\ ]\!]$ is a kell-m process. $\mathcal{A}[\![\ H[U]\ ]\!]$ has the form $\mathbf{0}$, $\mathcal{A}[\![\ K[\mathbf{0}]\ ]\!]$, **new** $c\ R$, $a(\widetilde{c}) \triangleright R$, or $L[x] \triangleright R$. We have already shown that advancing the execution of such an expression, composed in parallel with another kell-m expression, is also a kell-m expression. Hence, the result of $\mathcal{A}[\![\ K[H[U] \mid T]\ ]\!]$ is a kell-m process.
  * $U(\widetilde{w})$, by $\mathcal{A}$'s definition $\mathcal{A}[\![\ K[U(\widetilde{w}) \mid T]\ ]\!] \stackrel{def}{=} \mathcal{A}[\![\ K[U_d\{\widetilde{w}/\widetilde{y}\} \mid T]\ ]\!]$ where $U(\widetilde{y}) \stackrel{def}{=} U_d$. By structural induction $\mathcal{A}[\![\ K[U_d\{\widetilde{w}/\widetilde{y}\} \mid T]\ ]\!]$ is a kell-m process.

Without loss of generality, $\mathcal{A}[\![\ K[T|R]\ ]\!]$ is also a kell-m process.

Since both $\mathcal{S}[\![\ K[Q], B, K[Q], \{K\}\ ]\!]$ and $\mathcal{I}[\![\ B, \mathcal{A}[\![\ K[Q]\ ]\!]\ ]\!]$ are a MMCπ, then their composition via the π-calculus choice operator is also a MMCπ. Therefore $\mathcal{I}[\![\ B, P\ ]\!]$ is a MMCπ. In the case of Q being the parallel composition of processes R and T, we have also shown $\mathcal{I}[\![\ B, \mathcal{A}[\![\ K[R|T]\ ]\!]\ ]\!]$ and $\mathcal{I}[\![\ B, \mathcal{A}[\![\ K[T|R]\ ]\!]\ ]\!]$ are MMCπ, therefore, $\mathcal{I}[\![\ B, K[Q]\ ]\!]$ is a MMCπ.

- $K[x]\triangleright Q$, by $\mathcal{I}$'s definition $\mathcal{I}[\![\ B, K[x]\triangleright Q\ ]\!] \stackrel{def}{=} \mathcal{I}[\![\ B, \mathcal{F}[\![\ K(x,\emptyset)\triangleright Q\ ]\!]\ ]\!]$. Because $\mathcal{F}$ only performs alpha-conversions, the kell-m process resulting of $\mathcal{F}[\![\ K(x,\emptyset)\triangleright Q\ ]\!]$ will have the form $K(x',\emptyset)\triangleright Q'$, and we have already shown that the result of $\mathcal{I}[\![\ B, K(x',\emptyset)\triangleright Q'\ ]\!]$ is a MMCπ, therefore $\mathcal{I}[\![\ B, P\ ]\!]$ is a MMCπ.

- $Q(\widetilde{w})$, by $\mathcal{I}$'s definition $\mathcal{I}[\![\ B, Q(\widetilde{w})\ ]\!] \stackrel{def}{=} \mathcal{I}[\![\ B, \mathcal{F}[\![\ Q_d\{\widetilde{w}/\widetilde{y}\}\ ]\!]\ ]\!]$ where $Q(\widetilde{y}) \stackrel{def}{=} Q_d$. Because, $\mathcal{F}$ only performs alpha-conversions, by structural induction, $\mathcal{I}[\![\ B, \mathcal{F}[\![\ Q_d\{\widetilde{w}/\widetilde{y}\}\ ]\!]\ ]\!]$ is a MMCπ, therefore $\mathcal{I}[\![\ P\ ]\!]$ is a MMCπ.

- $Q \mid R$, by definition $\mathcal{I}[\![\ B, Q|R\ ]\!] \stackrel{def}{=} \mathcal{I}[\![\ B,\ Q\ ]\!] \mid \mathcal{I}[\![\ B,\ R\ ]\!]$. By induction both $\mathcal{I}[\![\ Q\ ]\!]$ and $\mathcal{I}[\![\ R\ ]\!]$ are MMCπs, and because the parallel composition of MMCπs is a MMCπ, then $\mathcal{I}[\![\ P\ ]\!]$ is a MMCπ.

We have shown that the result of the interpretation of a kell-m process is a MMC π-calculus process.

B.2 P is Indistinguishable from $\mathcal{I}[\![\ \{\},\ P\ ]\!]$

We want to show that when a kell-m process P transitions to Q, its encoding $\mathcal{I}[\![\ B, P\ ]\!]$, transitions to Q's encoding: if $P \stackrel{\alpha}{\rightarrow} Q$, then $\mathcal{I}[\![\ B, P\ ]\!] \stackrel{\alpha'}{\rightarrow} \mathcal{I}[\![\ B', Q\ ]\!]$.

As shown in Table B.1, the names involved in the actions α and α' are the same, but the actual parameters of the actions may differ.

$\widetilde{w}'$ is $\widetilde{w}$ with higher-order expressions replaced by higher-order indicators, and h_P is the higher-order indicator associated with P. The sets and variables used to expose kell containment information and bound names are included in the MMCπ actions. Because the names of the channels in communications do not change, an observer will not notice a change in the visibility predicates between P and its encoding, nor between Q and its encoding.

We start by defining in Figure B.1 $\mathcal{V}_M$, the visibility predicates for MMC. We do not specify a visibility predicate for $[a = b]P$ since such expressions are not generated by our

α	α'	
τ	$\tau(concr, K_a, K_c)$	with *concr* concretion, K_a, K_c kell cont. sets
$\overline{a}(\widetilde{w})$	$\overline{a}(\widetilde{w}', K_a, K_c, B)$	with K_a variable, K_c kell cont. set
$a(\widetilde{c})$	$a(\widetilde{c}, K_a, K_c, bnd)$	with K_a kell cont. set, K_c variable
$\overline{K}[P]$	$\overline{K}(\mathbf{new}\ h_P\ , K_a, K_c, B)$	with K_a variable, K_c kell cont. set
$K[x]$	$K(x, K_a, K_c, bnd)$	with K_a kell cont. set, K_c variable

Table B.1: Kell-m and Corresponding MMCπ Actions

$$
\begin{array}{lcl}
\mathcal{V}_M[\![\ \mathbf{0}\ ]\!] & \stackrel{def}{=} & \{\} \\
\mathcal{V}_M[\![\ \mathbf{new}\ a\ P\ ]\!] & \stackrel{def}{=} & \mathcal{V}_M[\![\ P\ ]\!] \setminus \{a\} \\
\mathcal{V}_M[\![\ \overline{a}(\widetilde{w}).P\ ]\!] & \stackrel{def}{=} & \{\overline{a}\} \\
\mathcal{V}_M[\![\ a(\widetilde{c}).P\ ]\!] & \stackrel{def}{=} & \{a\} \\
\mathcal{V}_M[\![\ P|Q\ ]\!] & \stackrel{def}{=} & \mathcal{V}_M[\![\ P\ ]\!] \cup \mathcal{V}_M[\![\ Q\ ]\!] \\
\mathcal{V}_M[\![\ P+Q\ ]\!] & \stackrel{def}{=} & \mathcal{V}_M[\![\ P\ ]\!] \cup \mathcal{V}_M[\![\ Q\ ]\!] \\
\mathcal{V}_M[\![\ \texttt{code(Op, P)}\ ]\!] & \stackrel{def}{=} & \mathcal{V}_M[\![\ P\ ]\!]
\end{array}
$$

Figure B.1: Visibility Predicates for MMCπ

encoding of kell-m processes. $\mathcal{V}$ the visibility for kell-m is defined in Figure 3.8, Section 3.5.

Using structural induction we will show the visibility predicates for P and its MMCπ encoding $\mathcal{I}[\![\ \{\}, P\ ]\!]$ are the same. For $\mathbf{0}$ and x the visibility predicates are trivially the same: both empty for P and $\mathcal{I}[\![\ \{\}, P\ ]\!]$. The other cases are:

- $\mathbf{new}\ a\ Q$, by the definition of $\mathcal{V}$, $\mathcal{V}_M$ and $\mathcal{I}$ we have $\mathcal{V}[\![\ \mathbf{new}\ a\ Q\ ]\!] \stackrel{def}{=} \mathcal{V}[\![\ Q\ ]\!] \setminus \{a\}$, $\mathcal{V}_M[\![\ \mathbf{new}\ a\ \mathcal{I}[\![\ B \cup \{a\}, Q\ ]\!]\ ]\!] \stackrel{def}{=} \mathcal{V}[\![\ \mathcal{I}[\![\ B \cup \{a\}, Q\ ]\!]\ ]\!] \setminus \{a\}$. By structural induction, $\mathcal{V}[\![\ Q\ ]\!] = \mathcal{V}[\![\ \mathcal{I}[\![\ B \cup \{a\}, Q\ ]\!]\ ]\!]$, therefore, $\mathcal{V}[\![\ \mathbf{new}\ a\ Q\ ]\!] = \mathcal{V}_M[\![\ \mathcal{I}[\![\ B, \mathbf{new}\ a\ Q\ ]\!]\ ]\!]$.

- $\overline{a}(\widetilde{w})$, by $\mathcal{I}$'s definition, $\mathcal{I}[\![\ B, \overline{a}(\widetilde{w})\ ]\!] \stackrel{def}{=} \mathit{Hwrite}(\overline{a}(\widetilde{w}, \emptyset), \mathbf{0}, B \setminus \{a\})$. Recall *Hwrite* replaces higher-order expressions in $\widetilde{w}$ with higher-order indicator names, and keeps track of those names in H. By $\mathcal{V}_M$'s and *Hwrite*'s definition: $\mathcal{V}_M[\![\ \mathit{Hwrite}(\overline{a}(\widetilde{w}, \emptyset), \mathbf{0}, B \setminus \{a\})\ ]\!] \stackrel{def}{=} \{\overline{a}\}$. Since, $\mathcal{V}[\![\ \overline{a}(\widetilde{w})\ ]\!] \stackrel{def}{=} \{\overline{a}\}$, then $\mathcal{V}[\![\ \overline{a}(\widetilde{w})\ ]\!] = \mathcal{V}_M[\![\ \mathcal{I}[\![\ B, \overline{a}(\widetilde{w})\ ]\!]\ ]\!]$.

- $a(\widetilde{c}) \triangleright Q$, by $\mathcal{V}$'s definition $\mathcal{V}[\![\ a(\widetilde{c}) \triangleright Q\ ]\!] \stackrel{def}{=} \{a\}$. By $\mathcal{V}_M$'s and $\mathcal{I}$'s definition:

$$\mathcal{I}[\![\ B, a(\widetilde{c}) \triangleright Q\ ]\!] \stackrel{def}{=} a(\widetilde{c}, K_s, K_c, bnd).\texttt{code}(\texttt{inst}(Q, \widetilde{c}, bnd, Q_\pi), Q_\pi)$$
$$\mathcal{V}[\![\ a(\widetilde{c}, bnd).\texttt{code}(\texttt{inst}(Q, \widetilde{c}, bnd, Q_\pi), Q_\pi)\ ]\!] \stackrel{def}{=} \{a\}$$

Then $\mathcal{V}[\![\ a(\widetilde{c}) \triangleright Q\ ]\!] = \mathcal{V}_M[\![\ a(\widetilde{c}, bnd).\mathtt{code}(\mathtt{inst}(Q, \widetilde{c}, bnd, Q_\pi), Q_\pi)\ ]\!]$

- $K[x] \triangleright Q$, by $\mathcal{V}$'s definition $\mathcal{V}[\![\ K[x] \triangleright Q\ ]\!] \stackrel{def}{=} \{K\}$. By $\mathcal{I}$'s and $\mathcal{V}_M$'s definitions:

$$\begin{aligned}
&\mathcal{V}_M[\![\ \mathcal{I}[\![\ B, K[x] \triangleright Q\ ]\!]\ ]\!] \\
&\quad = \mathcal{V}_M[\![\ \mathcal{I}[\![\ B, \mathcal{F}[\![\ K(x) \triangleright P\ ]\!]\ ]\!]\ ]\!] \\
&\quad = \mathcal{V}_M[\![\ K(x,\mathrm{bnd}).\mathtt{code}(\mathtt{inst}(Q, x, bnd, Q_\pi), Q_\pi)\ ]\!] \\
&\quad = \{K\} \\
&\quad = \mathcal{V}[\![\ K[x] \triangleright Q\ ]\!]
\end{aligned}$$

- $Q(\widetilde{w})$, by $\mathcal{V}$'s definition $\mathcal{V}[\![\ Q(\widetilde{w})\ ]\!] \stackrel{def}{=} \mathcal{V}[\![\ Q_d\{\widetilde{w}/\widetilde{y}\}\ ]\!]$, where $Q(\widetilde{y}) \stackrel{def}{=} Q_d$. By structural induction, $\mathcal{V}[\![\ Q_d\{\widetilde{w}/\widetilde{y}\}\ ]\!] = \mathcal{V}_M[\![\ \mathcal{I}[\![\ B, \mathcal{F}[\![\ Q_d\{\widetilde{w}/\widetilde{y}\}\ ]\!]\ ]\!]\ ]\!]$.

- $K[Q]$, by $\mathcal{V}$'s definition, $\mathcal{V}[\![\ K[Q]\ ]\!] \stackrel{def}{=} \{\overline{K}\} \cup \mathcal{V}[\![\ Q\ ]\!]$. By $\mathcal{I}$'s definition $\mathcal{I}[\![\ B, K[Q]\ ]\!] \stackrel{def}{=} \mathcal{S}[\![\ \mathbf{0}, B, K[Q], \emptyset\ ]\!] + \mathcal{I}[\![\ B, \mathcal{A}[\![\ K[Q]\ ]\!]\ ]\!]$ if Q is not the parallel composition of processes R and T; otherwise,$\mathcal{I}[\![\ B, K[Q]\ ]\!] \stackrel{def}{=} \mathcal{S}[\![\ \mathbf{0}, B, K[Q], \emptyset\ ]\!] + \mathcal{I}[\![\ B, \mathcal{A}[\![\ K[R|T]\ ]\!]\ ]\!] + \mathcal{I}[\![\ B, \mathcal{A}[\![\ K[T|R]\ ]\!]\ ]\!]$. We need to show that when Q is not $R|T$:

$$\begin{aligned}
&\mathcal{V}_M[\![\ \mathcal{S}[\![\ \mathbf{0}, B, K[Q], \emptyset\ ]\!]\ +\ \mathcal{I}[\![\ B, \mathcal{A}[\![\ K[Q]\ ]\!]\ ]\!]\ ]\!] \stackrel{def}{=} \\
&\mathcal{V}_M[\![\ \mathcal{S}[\![\ \mathbf{0}, B, K[Q], \emptyset\ ]\!]\ ]\!] \cup \mathcal{V}_M[\![\ \mathcal{I}[\![\ B, \mathcal{A}[\![\ K[Q]\ ]\!]\ ]\!]\ ]\!] = \\
&\{\overline{K}\} \cup \mathcal{V}[\![\ Q\ ]\!]
\end{aligned}$$

If $Q \equiv R|T$, we need to show:

$$\begin{aligned}
&\mathcal{V}_M[\![\ \mathcal{S}[\![\ \mathbf{0}, B, K[R|T], \emptyset\ ]\!]\ +\ \mathcal{I}[\![\ B, \mathcal{A}[\![\ K[R|T]\ ]\!]\ ]\!]\ +\ \mathcal{I}[\![\ B, \mathcal{A}[\![\ K[T|R]\ ]\!]\ ]\!]\ ]\!] \stackrel{def}{=} \\
&\mathcal{V}_M[\![\ \mathcal{S}[\![\ \mathbf{0}, B, K[R|T], \emptyset\ ]\!]\ ]\!]\ \cup\ \mathcal{V}_M[\![\ \mathcal{I}[\![\ B, \mathcal{A}[\![\ K[R|T]\ ]\!]\ ]\!]\ ]\!]\ \cup \\
&\quad \mathcal{V}_M[\![\ \mathcal{I}[\![\ B, \mathcal{A}[\![\ K[T|R]\ ]\!]\ ]\!]\ ]\!]\ =\ \{\overline{K}\} \cup \mathcal{V}[\![\ T|R\ ]\!]
\end{aligned}$$

Notice:

$$\begin{aligned}
&\mathcal{V}_M[\![\ \mathcal{S}[\![\ \mathbf{0}, B, K[Q], \emptyset\ ]\!]\ ]\!] \stackrel{def}{=} \\
&\mathcal{V}_M[\![\ \mathit{Hwrite}(\overline{K}(Q, \emptyset), \mathcal{I}[\![\ B, \mathbf{0}\{\mathbf{0}/K[Q]\}\ ]\!], B \setminus \{K\})\ +\ \mathcal{S}[\![\ K[Q], B, Q, \{K\}\ ]\!]\ ]\!] = \\
&\mathcal{V}_M[\![\ \mathit{Hwrite}(\overline{K}(Q, \emptyset), \mathcal{I}[\![\ B, \mathbf{0}\ ]\!], B \setminus \{K\})\ ]\!] \cup \mathcal{V}_M[\![\ \mathcal{S}[\![\ K[Q], B, Q, \{K\}\ ]\!]\ ]\!] = \\
&\mathcal{V}_M[\![\ \mathit{Hwrite}(\overline{K}(Q, \emptyset), \mathbf{0}, B \setminus \{K\})\ ]\!] \cup \mathcal{V}_M[\![\ \mathcal{S}[\![\ K[Q], B, Q, \{K\}\ ]\!]\ ]\!] = \\
&\{\overline{K}\} \cup \mathcal{V}_M[\![\ \mathcal{S}[\![\ K[Q], B, Q, \{K\}\ ]\!]\ ]\!]
\end{aligned}$$

Assuming Q is not $R|T$, left to show, is then:

$$\{\overline{K}\} \cup \mathcal{V}_M[\![\ \mathcal{S}[\![\ K[Q], B, Q, \{K\}\ ]\!]\ ]\!] \cup \mathcal{V}_M[\![\ \mathcal{I}[\![\ B, \mathcal{A}[\![\ K[Q]\ ]\!]\ ]\!]\ ]\!] = \{\overline{K}\} \cup \mathcal{V}[\![\ Q\ ]\!]$$

If Q is not a kell, then, by $\mathcal{S}$'s definition, $\mathcal{V}_M[\![\ \mathcal{S}[\![\ K[Q], B, Q, \{K\}\ ]\!]\ ]\!] \stackrel{def}{=} \mathcal{V}_M[\![\ \mathbf{0}\ ]\!] \stackrel{def}{=} \{\}$. So, for such a Q, we need only to show: $\mathcal{V}_M[\![\ \mathcal{I}[\![\ B, \mathcal{A}[\![\ K[Q]\ ]\!]\ ]\!]\ ]\!] = \mathcal{V}[\![\ Q\ ]\!]$. The possible cases for Q are:

– $\mathbf{0}$, then, $\mathcal{A}[\![\ K[Q]\ ]\!] = \mathbf{0}$, and therefore, $\mathcal{V}_M[\![\ \mathcal{I}[\![\ B, \mathcal{A}[\![\ K[Q]\ ]\!]\ ]\!]\ ]\!] = \mathcal{V}_M[\![\ \mathcal{I}[\![\ B, \mathbf{0}\ ]\!]\ ]\!] = \{\} = \mathcal{V}[\![\ Q\ ]\!]$.

– $\overline{a}(\widetilde{w})$, then, $\mathcal{A}[\![\ K[Q]\ ]\!] = \overline{a}(\widetilde{w}, \{K\})$, and

$$\begin{array}{rcl}
\mathcal{V}_M[\![\ \mathcal{I}[\![\ B, \mathcal{A}[\![\ K[Q]\ ]\!]\ ]\!]\ ]\!] = & \mathcal{V}_M[\![\ \mathcal{I}[\![\ B, \overline{a}(\widetilde{w}, \{K\})\ ]\!]\ ]\!] \\
= & \mathcal{V}_M[\![\ \mathcal{I}[\![\ B, \overline{a}(\widetilde{w}, \{K\})\ ]\!]\ ]\!] \\
= & \mathcal{V}_M[\![\ \mathit{Hwrite}(\overline{a}(\widetilde{w}, \{K\}), \mathbf{0}, B \setminus \{a\})\ ]\!] \\
= & \{\overline{a}\} \\
= & \mathcal{V}[\![\ \overline{a}(\widetilde{w})\ ]\!]
\end{array}$$

– $a(\widetilde{c}) \triangleright R$, then $\mathcal{A}[\![\ K[Q]\ ]\!] = a(\widetilde{c}, \{K\}) \triangleright K[R]$, and

$$\begin{array}{rcl}
\mathcal{V}_M[\![\ \mathcal{I}[\![\ B, \mathcal{A}[\![\ K[Q]\ ]\!]\ ]\!]\ ]\!] & = & \mathcal{V}_M[\![\ \mathcal{I}[\![\ B, a(\widetilde{c}, \{K\}) \triangleright K[R]\ ]\!]\ ]\!] \\
& = & \mathcal{V}_M[\![\ a(\widetilde{c}, \{K\}, K_c, \mathit{bnd}). \\
& & \qquad \mathtt{code}(\mathtt{inst}(R, \widetilde{c}, \mathit{bnd}, R_\pi), R_\pi)\ ]\!] \\
& = & \{a\} \\
& = & \mathcal{V}[\![\ a(\widetilde{c}) \triangleright R\ ]\!]
\end{array}$$

– $L[x] \triangleright R$, then $\mathcal{A}[\![\ K[Q]\ ]\!] = L(x, \{K\}) \triangleright K[R]$, and

$$\begin{array}{rcl}
\mathcal{V}_M[\![\ \mathcal{I}[\![\ B, \mathcal{A}[\![\ K[Q]\ ]\!]\ ]\!]\ ]\!] & = & \mathcal{V}_M[\![\ \mathcal{I}[\![\ B, L(x, \{K\}) \triangleright K[R]\ ]\!]\ ]\!] \\
& = & \mathcal{V}_M[\![\ L(x, \{K\}, K_c, \mathit{bnd}). \\
& & \qquad \mathtt{code}(\mathtt{inst}(R, x, \mathit{bnd}, R_\pi), R_\pi)\ ]\!] \\
& = & \{L\} \\
& = & \mathcal{V}[\![\ L[x] \triangleright R\ ]\!]
\end{array}$$

– $R(\widetilde{w})$, and $\mathcal{A}[\![\ R(\widetilde{w})\ ]\!] \stackrel{def}{=} \mathcal{A}[\![\ R_d(\{\widetilde{w}/\widetilde{y}\})\ ]\!]$, where $R(\widetilde{y}) \stackrel{def}{=} R_d$. By structural induction, we assume that:

$$\begin{array}{l}
\mathcal{V}_M[\![\ \mathcal{I}[\![\ B, \mathcal{A}[\![\ R(\widetilde{w})\ ]\!]\ ]\!]\ ]\!] \\
\quad = \mathcal{V}_M[\![\ \mathcal{I}[\![\ B, \mathcal{A}[\![\ R_d(\{\widetilde{w}/\widetilde{y}\})\ ]\!]\ ]\!]\ ]\!] \\
\quad = \mathcal{V}[\![\ R_d(\{\widetilde{w}/\widetilde{y}\})\ ]\!]
\end{array}$$

– **new** $a\ R$, then $\mathcal{A}[\![\ K[\mathbf{new}\ \mathrm{a}\ \ R]\ ]\!] = \mathbf{new}\ a\ \mathcal{A}[\![\ K[R]\ ]\!]$; assuming that R is not of the form **new** c ..., we have:

$$\begin{array}{rcl}
\mathcal{V}_M[\![\ \mathcal{I}[\![\ B, \mathcal{A}[\![\ K[\mathbf{new}\ \mathrm{a}\ \ R]\ ]\!]\ ]\!]\ ]\!] = & \mathcal{V}_M[\![\ \mathcal{I}[\![\ B, \mathbf{new}\ a\ \mathcal{A}[\![\ K[R]\ ]\!]\ ]\!]\ ]\!] \\
= & \mathcal{V}_M[\![\ \mathbf{new}\ a\ \mathcal{I}[\![\ B \cup \{a\}, \mathcal{A}[\![\ K[R]\ ]\!]\ ]\!]\ ]\!]
\end{array}$$

Assuming R is not a process with a kell, nor a parallel composition of kell-m processes, we have already shown $\mathcal{V}_M[\![\ \mathcal{I}[\![\ B, \mathcal{A}[\![\ K[R]\ ]\!]\ ]\!]\ ]\!] = \mathcal{V}[\![\ R\ ]\!]$ in these

cases, then:

$$\begin{aligned}
&\mathcal{V}_M[\![\ \mathcal{I}[\![\ B, \mathcal{A}[\![\ K[\textbf{new a}\ \ R]\ ]\!]\ ]\!]\ ]\!] \\
&\quad = (\{\overline{K}\} \cup \mathcal{V}_M[\![\ \mathcal{I}[\![\ B \cup \{a\}, \mathcal{A}[\![\ K[R]\ ]\!]\ ]\!]\ ]\!]) \setminus \{a\} \\
&\quad = (\{\overline{K}\} \cup \mathcal{V}[\![\ R\ ]\!]) \setminus \{a\} \\
&\quad = \{\overline{K}\} \cup \mathcal{V}[\![\ \textbf{new}\ a\ R\ ]\!]
\end{aligned}$$

When Q is the parallel composition of processes R and T, and these processes are not kells, it can also be shown the kell passivation code S generates the null process $\mathbf{0}$, and the only interesting remaining process is the advance-kell execution process. In such a case, showing that $\mathcal{V}_M[\![\ \mathcal{I}[\![\ B, \mathcal{A}[\![\ K[R|T]\ ]\!]\ ]\!]\ ]\!] \cup \mathcal{V}_M[\![\ \mathcal{I}[\![\ B, \mathcal{A}[\![\ K[T|R]\ ]\!]\ ]\!]\ ]\!] = \mathcal{V}[\![\ Q\ ]\!]$, requires computing the visibility predicates for the composed processes in Q. The procedure to follow is similar to the one just followed, and hence we skip it.

When Q has the form **new** a_1 **new** a_2 ...**new** a_n R, the restricted names are lifted from within the kell, and the resulting visibility expression is:

$$\begin{aligned}
&\mathcal{V}_M[\![\ \mathcal{I}[\![\ B, \mathcal{A}[\![\ K[\textbf{new}\ a_1, ..., a_n\ \ R]\ ]\!]\ ]\!]\ ]\!] \\
&\quad = \mathcal{V}_M[\![\ \mathcal{I}[\![\ B, \textbf{new}\ a1, ..., a_n\ \mathcal{A}[\![\ K[R]\ ]\!]\ ]\!]\ ]\!] \\
&\quad = \mathcal{V}_M[\![\ \textbf{new}\ a_1, ..., a_n\ \mathcal{I}[\![\ B \cup \textbf{new}\ a_1, ..., a_n\ , \mathcal{A}[\![\ K[R]\ ]\!]\ ]\!]\ ]\!] \\
&\quad = (\{\overline{K}\} \cup \mathcal{V}[\![\ R\ ]\!]) \setminus \{a_1, ..., a_n\} \\
&\quad = \{\overline{K}\} \cup \mathcal{V}[\![\ \textbf{new}\ a_1, ..., a_n\ R\ ]\!]
\end{aligned}$$

We still need to show that when Q is itself a kell, then

$$\{\overline{K}\} \cup \mathcal{V}_M[\![\ \mathcal{S}[\![\ K[Q], B, Q, \{K\}\ ]\!]\ ]\!] \cup \mathcal{V}_M[\![\ \mathcal{I}[\![\ B, \mathcal{A}[\![\ K[Q]\ ]\!]\ ]\!]\ ]\!] = \{\overline{K}\} \cup \mathcal{V}[\![\ Q\ ]\!]$$

We argue that in the case of nested kells, the suspend kell $\mathcal{S}$ definition generates null processes for each non-kell subprocess, and a concretion for each subkell, plus the result of advancing the execution of the kell. We will see that the resulting visibility predicates match. Let us assume $P = K[K_1[...[K_n[R]]...]]$, with R a non-kell process (i.e., R has a form different than $L[\ .]$). In such a process, we have $Q = K_1[...[K_n[R]]...]$, and:

$$\begin{aligned}
&\{\overline{K}\}\ \cup\ \mathcal{V}_M[\![\ \mathcal{S}[\![\ K[Q], B, Q, \{K\}\ ]\!]\ ]\!]\ \cup\ \mathcal{V}_M[\![\ \mathcal{I}[\![\ B, \mathcal{A}[\![\ K[Q]\ ]\!]\ ]\!]\ ]\!] = \\
&\{\overline{K}\}\ \cup\ \mathcal{V}_M[\![\ \mathit{Hwrite}(\overline{K_1}(K_2[K_3[...[K_n[R]]...]], \{K\}).\mathcal{I}[\![\ B, K[\mathbf{0}]\ ]\!], B \setminus \{K, K_1\})\ ]\!] \\
&\qquad \cup\ \mathcal{V}_M[\![\ \mathit{Hwrite}(\overline{K_2}(K_3[...[K_n[R]]...], \{K, K_1\}). \\
&\qquad\qquad \mathcal{I}[\![\ B, K[K_1[\mathbf{0}]]\ ]\!], B \setminus \{K, K_1, K_2\})\ ]\!] \\
&\qquad \cup\ \cdots \\
&\qquad \cup\ \mathcal{V}_M[\![\ \mathit{Hwrite}(\overline{K_n}(R, \{K, K_1, K_2, ..., K_{n-1}\}). \\
&\qquad\qquad \mathcal{I}[\![\ B, K[K_1[K_2[...[K_{n-1}[\mathbf{0}]]...]]]\ ]\!], B \setminus \{K, K_1, ..., K_n\})\ ]\!] \\
&\qquad \cup\ \mathcal{V}_M[\![\ \mathcal{I}[\![\ B, \mathcal{A}[\![\ K[K_1[...[K_n[R]]...]]\ ]\!]\ ]\!]\ ]\!] =
\end{aligned}$$

$$\{\overline{K}\} \cup \{\overline{K_1}\} \cup \{\overline{K_2}\} \cup \cdots \cup \{\overline{K_n}\} \cup \mathcal{V}_M[\![\mathcal{I}[\![B, K[K_1[...[\mathcal{A}[\![K_n[R]]\!]...]]]\!]]\!]$$

The expression $\mathcal{I}[\![B, K[K_1[...[\mathcal{A}[\![K_n[R]]\!]...]]]\!]$, by $\mathcal{I}$'s and $\mathcal{S}$'s definitions, takes the result of $\mathcal{A}[\![K_n[R]]\!]$ and, on each recursive invocation of $\mathcal{A}$, moves the expression one kell outwards. For example, if $R = a(c) \triangleright R'$:

$\mathcal{A}$ inv.	Expression
0	$\mathcal{A}[\![K[\mathcal{A}[\![K_1[\mathcal{A}[\![K_2[...[\mathcal{A}[\![K_{n-1}[\mathcal{A}[\![K_n[a(c) \triangleright R']]\!]]\!]...]]\!]]\!]]\!]]\!]$
1	$\mathcal{A}[\![K[\mathcal{A}[\![K_1[\mathcal{A}[\![K_2[...[\mathcal{A}[\![K_{n-1}[a(c, \{K_n\}) \triangleright K_n[R']]]\!]...]]\!]]\!]]\!]]\!]$
2	$\mathcal{A}[\![K[\mathcal{A}[\![K_1[\mathcal{A}[\![K_2[...[a(c, \{K_n, K_{n-1}\}) \triangleright K_{n-1}[K_n[R']]]...]]\!]]\!]]\!]]\!]$
...	...
$n+1$	$a(c, \{K_n, K_{n-1}, ..., K_1, K\}) \triangleright K[K_1[K_2[...[K_n[R']]...]]]$

Therefore:

$$\begin{aligned}&\mathcal{V}_M[\![\mathcal{I}[\![B, K[K_1[...[\mathcal{A}[\![K_n[R]]\!]...]]]\!]]\!] \\ &\quad = \{K, K_1, K_2, ..., K_n\} \cup \mathcal{V}_M[\![\mathcal{A}[\![K_n[R]]\!]]\!]\end{aligned}$$

Finally, by $\mathcal{V}$'s definition:

$$\mathcal{V}[\![K_1[K_2[...[K_n[R]]...]]]\!] = \{K_1, K_2, ..., K_n\} \cup \mathcal{V}[\![R]\!]$$

We have previously shown, for non-kell processes R, $\mathcal{V}_M[\![\mathcal{I}[\![B, \mathcal{A}[\![K_n[R]]\!]]\!]]\!] = \mathcal{V}[\![R]\!]$. Now, we have all the pieces:

$$\begin{aligned}&\{\overline{K}\} \cup \mathcal{V}_M[\![\mathcal{S}[\![K[Q], B, Q, \{K\}]\!]]\!] \cup \mathcal{V}_M[\![\mathcal{I}[\![B, \mathcal{A}[\![K[Q]]\!]]\!]]\!] \\ &\quad = \{\overline{K}\} \cup \{K, K_1, ..., K_n\} \cup \mathcal{V}_M[\![\mathcal{A}[\![K_n[R]]\!]]\!] \\ &\quad = \{K, K_1, ..., K_n\} \cup \mathcal{V}_M[\![\mathcal{A}[\![K_n[R]]\!]]\!] \\ &\quad = \{K\} \cup \mathcal{V}[\![K_1[K_2[...[K_n[R]]...]]]\!]\end{aligned}$$

With this, the last case for $K[Q]$, we have shown that $\mathcal{V}[\![K[Q]]\!] = \mathcal{V}_M[\![\mathcal{I}[\![B, K[Q]]\!]]\!]$.

- $Q|R$, then $\mathcal{V}[\![P|Q]\!] \stackrel{def}{=} \mathcal{V}[\![P]\!] \cup \mathcal{V}[\![Q]\!]$. By $\mathcal{I}$'s and $\mathcal{V}_M$'s definitions, we have $\mathcal{I}[\![B, Q|R]\!] \stackrel{def}{=} \mathcal{I}[\![B, Q]\!] \,|\, \mathcal{I}[\![B, R]\!]$, and $\mathcal{V}_M[\![\mathcal{I}[\![B, Q]\!] | \mathcal{I}[\![B, R]\!]]\!] \stackrel{def}{=} \mathcal{V}_M[\![\mathcal{I}[\![B, Q]\!]]\!] \cup \mathcal{V}_M[\![\mathcal{I}[\![B, R]\!]]\!]$. Finally, by structural induction, $\mathcal{V}[\![Q]\!] = \mathcal{V}_M[\![\mathcal{I}[\![B, Q]\!]]\!]$, and $\mathcal{V}[\![R]\!] = \mathcal{V}_M[\![\mathcal{I}[\![B, R]\!]]\!]$; therefore, $\mathcal{V}[\![P|Q]\!] = \mathcal{V}_M[\![\mathcal{I}[\![B, Q|R]\!]]\!]$.

We have shown that, initially, P and its encoding $\mathcal{I}[\![\{\}, P]\!]$ have the same visibility predicates. By using LTS semantics, we will now show that if $P \to P'$, then $\mathcal{I}[\![B, P]\!] \to \mathcal{I}[\![B'', P'']\!]$, with P'' structurally equivalent to P': $P'' \equiv P'$. For the kell-m calculus we will use the LTS semantics specified in Section 3.4.1. For MMCπ we use the LTS semantics from Section B.1. We will only show the L-* version of the transition rules.

In the discussion below, K_c and K_a may be used for kell containment sets and for variables. If an abstraction K_a is the kell containment set for the abstraction and K_c is a variable, to be instantiated with the kell containment set of a matching concretion in τ transitions. In concretions K_c is the kell containment set of the concretion and K_a is the variable.

- OUT. The encoding for $\overline{a}(\widetilde{w})$ is:

$$\begin{aligned}\mathcal{I}[\![\ B, \overline{a}(\widetilde{w})\ ]\!] &\stackrel{def}{=} Hwrite(\overline{a}(\widetilde{w}, \emptyset), \mathbf{0}, B \setminus \{a\}) \\ &\equiv \mathbf{new}\ \widetilde{h}\ \overline{a}(\widetilde{w}', K_a, \emptyset, B \setminus \{a\}).\mathbf{0}\end{aligned}$$

  With $\widetilde{h}$ the higher-order indicators for the higher-order expressions in $\widetilde{w}$. $\widetilde{w}'$ is $\widetilde{w}$ with higher-order expressions replaced by their higher-order indicators $h_i \in \widetilde{h}$. The kell-m transition is:

$$\overline{a}(\widetilde{w}) \xrightarrow{\overline{a}(\widetilde{w})} \mathbf{0}$$

  πPRE is the corresponding MMCπ transition:

$$\mathbf{new}\ \widetilde{h}\ \overline{a}(\widetilde{w}', K_a, \emptyset, B \setminus \{a\}) \xrightarrow{0,\overline{a}(\widetilde{w}'', K_a, \emptyset, B')} \mathbf{0}$$

  $\widetilde{w}''$ is $\widetilde{w}'$ with higher-order indicators h_j replaced by **new** h_j; and B' is $B \setminus \{a\}$ with all $b_i \in B \setminus \{a\}$ replaced by **new** b_i. Therefore, trivially:

$$\mathcal{I}[\![\ B, \overline{a}(\widetilde{w})\ ]\!] \xrightarrow{0,\overline{a}(\widetilde{w}'', K_a, \emptyset, B')} \mathcal{I}[\![\ B, \mathbf{0}\ ]\!]$$

- IN. The encoding for $a(\widetilde{c}) \triangleright Q$ is:

$$\begin{aligned}\mathcal{I}[\![\ B, a(\widetilde{c}) \triangleright Q\ ]\!] &\stackrel{def}{=} a(\widetilde{c}, \emptyset, K_c, bnd).\mathtt{code}(\mathtt{inst}(Q, \widetilde{c}, bnd, Q_\pi), Q_\pi) \\ &\equiv a(\widetilde{c}, bnd).\mathcal{I}[\![\ B \cup bnd, \mathcal{F}[\![\ \mathcal{H}[\![\ Q\ ]\!]\ ]\!]\ ]\!]\end{aligned}$$

  The result of the kell-m transitions is:

$$a(\widetilde{x}) \triangleright Q \xrightarrow{a(\widetilde{c})} Q$$

  The corresponding MMCπ transition is, once again, πPRE:

$$\begin{aligned}&a(\widetilde{c}, \emptyset, K_c, bnd).\mathcal{I}[\![\ B \cup bnd, \mathcal{F}[\![\ \mathcal{H}[\![\ Q\ ]\!]\ ]\!]\ ]\!] \xrightarrow{0,a(\widetilde{c},K_c,\emptyset,bnd)} \\ &\quad \mathcal{I}[\![\ B \cup bnd, \mathcal{F}[\![\ \mathcal{H}[\![\ Q\ ]\!]\ ]\!]\ ]\!]\end{aligned}$$

  Since there is no communication yet, $\mathcal{H}[\![\ Q\ ]\!] = Q$. $\mathcal{F}$ is used to avoid unintended capturing of names in process expressions. The result of $\mathcal{F}$ on a process is structurally equivalent to the original process, therefore, $\mathcal{F}[\![\ \mathcal{H}[\![\ Q\ ]\!]\ ]\!] \equiv Q$, and:

$$\mathcal{I}[\![\ B, a(\widetilde{c}) \triangleright Q\ ]\!] \xrightarrow{0,a(\widetilde{c},\emptyset,K_c,bnd)} \mathcal{I}[\![\ B \cup bnd, Q\ ]\!]$$

- KellOut. The encoding for $K[Q]$, assuming Q is not the parallel composition of processes T and R, is:

$$\begin{aligned}\mathcal{I}[\![\ B, K[Q]\ ]\!] &\stackrel{def}{=} \mathcal{S}[\![\ \mathbf{0}, B, K[Q], \emptyset\ ]\!] + \mathcal{I}[\![\ B, \mathcal{A}[\![\ K[Q]\ ]\!]\ ]\!] \\ &\equiv (\mathbf{new}\ h_Q\ \overline{K}(h_Q, K_a, \emptyset, B \setminus \{K\}).\mathbf{0}) + \\ &\quad \mathcal{S}[\![\ K[Q], B, Q, \{K\}\ ]\!]\ +\ \mathcal{I}[\![\ B, \mathcal{A}[\![\ K[Q]\ ]\!]\ ]\!]\end{aligned}$$

Where h_Q is the higher-order indicator for Q. The transition for kell-m is:

$$K[Q] \xrightarrow{\overline{K}[Q]} \mathbf{0}$$

Notice, in MMCπ, $\overline{K}(h)$ is a concretion, and by πSum:

$$\begin{aligned}&(\mathbf{new}\ h_Q\ \overline{K}(h_Q, K_a, \emptyset, B \setminus \{K\}).\mathbf{0})\ +\ \mathcal{S}[\![\ K[Q], B, Q, \{K\}\ ]\!] + \\ &\mathcal{I}[\![\ B, \mathcal{A}[\![\ K[Q]\ ]\!]\ ]\!] \xrightarrow{0, \overline{K}(\mathbf{new}\ h_Q\ , K_a, \emptyset, B')} \mathbf{0}\end{aligned}$$

B' is $B \setminus \{K\}$, with $b_i \in B \setminus \{K\}$ replaced by $\mathbf{new}\ b_i$. Since $\mathcal{I}[\![\ B, \mathbf{0}\ ]\!] \equiv \mathbf{0}$, then:

$$\mathcal{I}[\![\ B, K[Q]\ ]\!] \xrightarrow{0, \overline{K}(\mathbf{new}\ h_Q\ , K_a, \emptyset, B')} \mathcal{I}[\![\ B, \mathbf{0}\ ]\!]$$

When $Q \equiv R|T$, we have:

$$\begin{aligned}\mathcal{I}[\![\ B, K[Q]\ ]\!] &\stackrel{def}{=} \mathcal{S}[\![\ \mathbf{0}, B, K[Q], \emptyset\ ]\!] + \mathcal{I}[\![\ B, \mathcal{A}[\![\ K[R|T]\ ]\!]\ ]\!] + \\ &\quad \mathcal{I}[\![\ B, \mathcal{A}[\![\ K[T|R]\ ]\!]\ ]\!] \\ &\equiv (\mathbf{new}\ h_Q\ \overline{K}(h_Q, K_a, \emptyset, B \setminus \{K\}).\mathbf{0}) + \mathcal{S}[\![\ K[Q], B, Q, \{K\}\ ]\!] + \\ &\quad \mathcal{I}[\![\ B, \mathcal{A}[\![\ K[R|T]\ ]\!]\ ]\!] + \mathcal{I}[\![\ B, \mathcal{A}[\![\ K[T|R]\ ]\!]\ ]\!]\end{aligned}$$

Again, by πSum,

$$\mathcal{I}[\![\ B, K[Q]\ ]\!] \xrightarrow{0, \overline{K}(\mathbf{new}\ h_Q\ K_a, \emptyset, B')} \mathcal{I}[\![\ B, \mathbf{0}\ ]\!]$$

- KellIn. The encoding for $K[x] \triangleright Q$ is:

$$\begin{aligned}\mathcal{I}[\![\ B, K[x] \triangleright Q\ ]\!] &\stackrel{def}{=} \mathcal{I}[\![\ B, K(x, \emptyset) \triangleright Q\ ]\!] \\ &\equiv K(x, \emptyset, K_c, bnd).\mathcal{I}[\![\ B \cup bnd, \mathcal{F}[\![\ \mathcal{H}[\![\ Q\ ]\!]\ ]\!]\ ]\!]\end{aligned}$$

The transition for kell-m is:

$$K[x] \xrightarrow{K[x]} Q$$

Using the πPre transition for MMCπ we obtain:

$$\begin{aligned}&K(x, \emptyset, K_c, bnd).\mathcal{I}[\![\ B \cup bnd, \mathcal{F}[\![\ \mathcal{H}[\![\ Q\ ]\!]\ ]\!]\ ]\!] \xrightarrow{0, K(x, \emptyset, K_c, bnd)} \\ &\quad \mathcal{I}[\![\ B \cup bnd, \mathcal{F}[\![\ \mathcal{H}[\![\ Q\ ]\!]\ ]\!]\ ]\!]\end{aligned}$$

Since no communication has taken place, $Q \equiv \mathcal{F}[\![\ \mathcal{H}[\![\ Q\ ]\!]\ ]\!]$, therefore,

$$\mathcal{I}[\![\ B, K[x] \triangleright Q\ ]\!] \xrightarrow{0, K(x, \emptyset, K_c, bnd)} \mathcal{I}[\![\ B \cup bnd, Q\ ]\!]$$

- RESTRICT. We have $Q \xrightarrow{\alpha} R$, and we will assume, by structural induction, $\mathcal{I}[\![\ B, Q\ ]\!] \xrightarrow{M,\alpha'} \mathcal{I}[\![\ B, R\ ]\!]$. We need to show that $\mathcal{I}[\![\ B, \mathbf{new}\ c\ Q\ ]\!] \xrightarrow{M,\alpha'} \mathcal{I}[\![\ B, \mathbf{new}\ c\ R\ ]\!]$, with $c \notin bn(\alpha')$. By $\mathcal{I}$'s definition:

$$\mathcal{I}[\![\ B, \mathbf{new}\ c\ Q\ ]\!] \stackrel{def}{=} \mathbf{new}\ c\ \mathcal{I}[\![\ B, Q\ ]\!]$$

By πRES, when $c \notin names(M, \alpha')$,

$$\mathbf{new}\ c\ \mathcal{I}[\![\ B, Q\ ]\!] \xrightarrow{M,\alpha'} \mathbf{new}\ c\ \mathcal{I}[\![\ B, R\ ]\!]$$

And, since $\mathbf{new}\ c\ \mathcal{I}[\![\ B, R\ ]\!] \equiv \mathcal{I}[\![\ B, \mathbf{new}\ c\ R\ ]\!]$, we have

$$\mathcal{I}[\![\ B, \mathbf{new}\ c\ Q\ ]\!] \xrightarrow{M,\alpha'} \mathcal{I}[\![\ B, \mathbf{new}\ c\ R\ ]\!]$$

- PAR. We have $Q \xrightarrow{\alpha} R$ and, by structural induction, assume $\mathcal{I}[\![\ B_Q, Q\ ]\!] \xrightarrow{M,\alpha'} \mathcal{I}[\![\ B_R, R\ ]\!]$. We need to show that $\mathcal{I}[\![\ B_Q, Q|Q'\ ]\!] \xrightarrow{M,\alpha'} \mathcal{I}[\![\ B_R, R|Q'\ ]\!]$ when $\alpha' \notin fn(Q')$. By πPAR:

$$\mathcal{I}[\![\ B_Q, Q\ ]\!] \mid \mathcal{I}[\![\ B_Q, Q'\ ]\!] \xrightarrow{M,\alpha'} \mathcal{I}[\![\ B_R, R\ ]\!] \mid \mathcal{I}[\![\ B_R, Q'\ ]\!]$$

By $\mathcal{I}'s$ definition, $\mathcal{I}[\![\ B_Q, Q\ ]\!] \mid \mathcal{I}[\![\ B_Q, Q'\ ]\!] \equiv \mathcal{I}[\![\ B_Q, Q|Q'\ ]\!]$. Therefore,

$$\mathcal{I}[\![\ B_Q, Q|Q'\ ]\!] \xrightarrow{M,\alpha'} \mathcal{I}[\![\ B_R, R|Q'\ ]\!]$$

- OPEN. We have $Q \xrightarrow{\overline{a}(\widetilde{w})} R$, and we have already shown:

$$\mathcal{I}[\![\ B_Q, Q\ ]\!] \xrightarrow{0,\overline{a}(\widetilde{w}'', K_a, K_c, B')} \mathcal{I}[\![\ B_R, R\ ]\!]$$

By πOPEN:

$$\mathbf{new}\ c\ \mathcal{I}[\![\ B_Q, Q\ ]\!] \xrightarrow{0,\overline{a}(\widetilde{u}, K_a, K_c, B')} \mathcal{I}[\![\ B_R, R\ ]\!]$$

With $\widetilde{u}$ as $\widetilde{w}''$, but with c replaced by $\mathbf{new}\ c$. By $\mathcal{I}$'s definition, we have $\mathbf{new}\ c\ \mathcal{I}[\![\ B_Q, Q\ ]\!] \equiv \mathcal{I}[\![\ B_Q, \mathbf{new}\ c\ Q\ ]\!]$, therefore,

$$\mathcal{I}[\![\ B_Q, \mathbf{new}\ c\ Q\ ]\!] \xrightarrow{0,\overline{a}(\widetilde{u}, K_a, K_c, B')} \mathcal{I}[\![\ B_R, R\ ]\!]$$

- L-REACT and L-CLOSE. We have $Q \xrightarrow{a(\widetilde{c})} Q'$ and $R \xrightarrow{\overline{a}(\widetilde{w})} R'$. By structural induction we assume,

$$\mathcal{I}[\![\ B_Q, Q\ ]\!] \xrightarrow{M,a(\widetilde{c}, K_a, K_c, bnd)} \mathcal{I}[\![\ B'_Q, Q'\ ]\!]$$

and,

$$\mathcal{I}[\![\ B_R, R\ ]\!] \xrightarrow{N,\overline{a}(\widetilde{w}'',K_a,K_c,B')} \mathcal{I}[\![\ B'_R, R'\ ]\!]$$

By πCom and πClose, depending on whether or not there are restricted names or higher-order indicators being passed in the communication, we have:

$$\begin{array}{l}\mathcal{I}[\![\ B_Q, Q\ ]\!] \mid \mathcal{I}[\![\ B_R, R\ ]\!] \\ \quad \xrightarrow{M\cup N,\tau} \textbf{new}\ \widetilde{d}\ (\mathcal{I}[\![\ B'_Q, Q'\ ]\!]\{(\widetilde{w}'', B')/(\widetilde{c}, bnd)\} \mid \mathcal{I}[\![\ B'_R, R'\ ]\!])\end{array}$$

where $\widetilde{d}$ are the restricted names being passed in the communication (if any). Finally, by $\mathcal{I}$'s definition, we have:

$$\begin{array}{l}\mathcal{I}[\![\ B_Q \cup B_R, Q|R\ ]\!] \\ \quad \xrightarrow{M\cup N,\tau} \mathcal{I}[\![\ B'_Q \cup B'_R, \textbf{new}\ \widetilde{d}\ (Q'\{(\widetilde{w}'', B')/(\widetilde{c}, bnd)\} \mid R')\ ]\!]\end{array}$$

Note by $\mathcal{I}$'s definition, $\mathcal{I}[\![\ B'_Q, Q'\ ]\!]$ has the form $\mathcal{I}[\![\ B'_Q, \mathcal{F}[\![\ \mathcal{H}[\![\ ...\ ]\!]\ ]\!]\ ]\!]$. This guarantees that, on communication, higher-order indicators are replaced by their associated higher-order expressions.

- L-Suspend. We have $Q \xrightarrow{K[x]} Q'$ and $R \xrightarrow{\overline{K}(T)} R'$, where T is the process inside kell K. By structural induction we assume,

$$\mathcal{I}[\![\ B_Q, Q\ ]\!] \xrightarrow{M,K(x,K_a,K_c,bnd)} \mathcal{I}[\![\ B'_Q, Q'\ ]\!]$$

And,

$$\mathcal{I}[\![\ B_R, R\ ]\!] \xrightarrow{N,\overline{K}(\textbf{new}\ h_T\ ,K_a,K_c,B')} \mathcal{I}[\![\ B'_R, R'\ ]\!]$$

Where h_T is the higher-order indicator for T. Since we use channels to represent kells, using πClose we obtain:

$$\begin{array}{l}\mathcal{I}[\![\ B_Q, Q\ ]\!] \mid \mathcal{I}[\![\ B_R, R\ ]\!] \\ \quad \xrightarrow{M\cup N,\tau} \textbf{new}\ h_T, \widetilde{d}\ (\mathcal{I}[\![\ B'_Q, Q'\ ]\!]\{(h_T, B')/(x, bnd)\} \mid \mathcal{I}[\![\ B'_R, R'\ ]\!])\end{array}$$

Where $\widetilde{d}$ are the restricted names, with the exception of h_T, being passed in the communication (if any). Using $\mathcal{I}$'s definition, we obtain:

$$\begin{array}{l}\mathcal{I}[\![\ B_Q \cup B_R, Q|R\ ]\!] \\ \quad \xrightarrow{M\cup N,\tau} \mathcal{I}[\![\ B'_Q \cup B'_R, \textbf{new}\ h_T, \widetilde{c}\ (Q'\{(h_T, B')/(x, bnd)\} \mid R')\ ]\!]\end{array}$$

As with rules L-React and L-Close, by $\mathcal{I}$'s definition, $\mathcal{I}[\![\ B'_Q, Q'\ ]\!]$ has the form $\mathcal{I}[\![\ B'_Q, \mathcal{F}[\![\ \mathcal{H}[\![\ ...\ ]\!]\ ]\!]\ ]\!]$, guaranteeing that the higher-order indicator h_T is replaced by its associated higher order expression T.

- ADVANCE. When $Q \xrightarrow{\alpha} R$, then $K[Q] \xrightarrow{\alpha} K[R]$. By structural induction, we assume

$$\mathcal{I}[\![\ B, Q\ ]\!] \xrightarrow{M,\alpha'} \mathcal{I}[\![\ B', R\ ]\!]$$

As we have seen,

$$\begin{aligned}\mathcal{I}[\![\ B, K[Q]\ ]\!] \equiv\ & \overline{K}(h_Q, K_a, \emptyset, B \setminus \{K\}).\mathbf{0} + \mathcal{S}[\![\ K[Q], B, Q, \{K\}\ ]\!] + \\ & \mathcal{I}[\![\ B, \mathcal{A}[\![\ K[Q]\ ]\!]\ ]\!]\end{aligned}$$

Or,

$$\begin{aligned}\mathcal{I}[\![\ B, K[Q]\ ]\!] \equiv\ & \overline{K}(h_Q, K_a, \emptyset, B \setminus \{K\}).\mathbf{0} + \mathcal{S}[\![\ K[Q], B, Q, \{K\}\ ]\!] + \\ & \mathcal{I}[\![\ B, \mathcal{A}[\![\ K[U|T]\ ]\!]\ ]\!] + \mathcal{I}[\![\ B, \mathcal{A}[\![\ K[T|U]\ ]\!]\ ]\!]\end{aligned}$$

If $Q \equiv U|T$. In general, Q can have one of the following forms:

- $a(\widetilde{c}) \triangleright R$:

$$\begin{aligned}&(\mathbf{new}\ h_Q\ \overline{K}(h_Q, K_a, \emptyset, B \setminus \{K\}).\mathbf{0}) + \mathcal{S}[\![\ K[Q], B, Q, \{K\}\ ]\!] + \\ &\mathcal{I}[\![\ B, \mathcal{A}[\![\ K[Q]\ ]\!]\ ]\!] \\ &\quad\equiv\ (\mathbf{new}\ h_Q\ \overline{K}(h_Q, K_a, \emptyset, B \setminus \{K\}).\mathbf{0}) + \mathcal{S}[\![\ K[Q], B, Q, \{K\}\ ]\!] + \\ &\qquad \mathcal{I}[\![\ B, a(\widetilde{c}) \triangleright K[R]\ ]\!] \\ &\quad\equiv\ (\mathbf{new}\ h_Q\ \overline{K}(h_Q, K_a, \emptyset, B \setminus \{K\}).\mathbf{0}) + \mathcal{S}[\![\ K[Q], B, Q, \{K\}\ ]\!] + \\ &\qquad a(\widetilde{c}, \{K\}, K_c, bnd).\mathcal{I}[\![\ B \cup bnd, \mathcal{F}[\![\ \mathcal{H}[\![\ \mathrm{K[R]}\ ]\!]\ ]\!]\ ]\!] \xrightarrow{0,a(\widetilde{c},\{K\},K_c,bnd)} \\ &\qquad \mathcal{I}[\![\ B \cup bnd, \mathcal{F}[\![\ \mathcal{H}[\![\ \mathrm{K[R]}\ ]\!]\ ]\!]\ ]\!]\end{aligned}$$

  Since no communication has yet taken place, $\mathcal{F}[\![\ \mathcal{H}[\![\ \mathrm{K[R]}\ ]\!]\ ]\!] \equiv K[R]$. Therefore,

$$\mathcal{I}[\![\ B, K[Q]\ ]\!] \xrightarrow{0,a(\widetilde{c},\{K\},K_c,bnd)} \mathcal{I}[\![\ B \cup bnd, K[R]\ ]\!]$$

- $\overline{a}(\widetilde{w})$. In this case $R = \mathbf{0}$, and

$$\begin{aligned}&(\mathbf{new}\ h_Q\ \overline{K}(h_Q, K_a, \emptyset, B \setminus \{K\}).\mathbf{0}) + \mathcal{S}[\![\ K[\overline{a}(\widetilde{w})], B, \overline{a}(\widetilde{w}), \{K\}\ ]\!] + \\ &\mathcal{I}[\![\ B, \mathcal{A}[\![\ K[\overline{a}(\widetilde{w})]\ ]\!]\ ]\!] \\ &\quad\equiv\ (\mathbf{new}\ h_Q\ \overline{K}(h_Q, K_a, \emptyset, B \setminus \{K\}).\mathbf{0}) + \mathcal{S}[\![\ K[\overline{a}(\widetilde{w})], B, \overline{a}(\widetilde{w}), \{K\}\ ]\!] + \\ &\qquad \mathcal{I}[\![\ B, \overline{a}(\widetilde{w})\ ]\!] \\ &\quad\equiv\ (\mathbf{new}\ h_Q\ \overline{K}(h_Q, K_a, \emptyset, B \setminus \{K\}).\mathbf{0}) + \mathcal{S}[\![\ K[\overline{a}(\widetilde{w})], B, \overline{a}(\widetilde{w}), \{K\}\ ]\!] + \\ &\qquad \mathbf{new}\ \widetilde{h}\ \overline{a}(\widetilde{w}', \emptyset, B \setminus \{a\}).\mathbf{0} \xrightarrow{0,\overline{a}(\widetilde{w}'',K_a,\{K\},B'')} \mathbf{0}\end{aligned}$$

  With $\widetilde{h}$ the higher-order indicators for the higher-order expressions in $\widetilde{w}$. $\widetilde{w}'$ is $\widetilde{w}$ with higher-order expressions replaced by their higher-order indicators $h_i \in \widetilde{h}$. $\widetilde{w}''$ is $\widetilde{w}'$, with the higher-order indicators, h_i, replaced with **new** h_i ; finally, B'' is $B \setminus \{a\}$ with all $b_i \in B \setminus \{a\}$ replaced by **new** b_i. Since $\mathcal{I}[\![\ B, \mathbf{0}\ ]\!] \equiv \mathbf{0}$:

$$\mathcal{I}[\![\ B, K[\overline{a}(\widetilde{w})]\ ]\!] \xrightarrow{0,\overline{a}(\widetilde{w}'',K_a,\{K\},B'')} \mathcal{I}[\![\ B \cup bnd, K[0]\ ]\!]$$

– $L[x] \triangleright T$. In this case $R = T$, and

$$\begin{array}{l}(\textbf{new}\ h_Q\ \overline{K}(h_Q, K_a, \emptyset, B \setminus \{K\}).\mathbf{0}) + \mathcal{S}[\![\ K[L[x] \triangleright T], B, L[x] \triangleright T\ ]\!] + \\ \mathcal{I}[\![\ B, \mathcal{A}[\![\ K[L[x] \triangleright T]\ ]\!]\ ]\!] \\ \quad \equiv\ (\textbf{new}\ h_Q\ \overline{K}(h_Q, K_a, \emptyset, B \setminus \{K\}).\mathbf{0}) + \\ \qquad \mathcal{S}[\![\ K[L[x] \triangleright T], B, L[x] \triangleright T, \{K\}\ ]\!] + \mathcal{I}[\![\ B, L(x) \triangleright K[T]\ ]\!] \\ \quad \equiv\ (\textbf{new}\ h_Q\ \overline{K}(h_Q, K_a, \emptyset, B \setminus \{K\}).\mathbf{0}) + \\ \qquad \mathcal{S}[\![\ K[L[x] \triangleright T], B, L[x] \triangleright T, \{K\}\ ]\!] + \\ \qquad L(x, \{K\}, K_c, bnd).\mathcal{I}[\![\ B \cup bnd, \mathcal{F}[\![\ \mathcal{H}[\![\ K[T]\ ]\!]\ ]\!]\ ]\!] \xrightarrow{0, L(x, \{K\}, K_c, bnd)} \\ \qquad \mathcal{I}[\![\ B \cup bnd, \mathcal{F}[\![\ \mathcal{H}[\![\ K[T]\ ]\!]\ ]\!]\ ]\!]\end{array}$$

Since no communication has yet taken place, $\mathcal{F}[\![\ \mathcal{H}[\![\ K[T]\ ]\!]\ ]\!] \equiv K[T]$, and:

$$\mathcal{I}[\![\ B, K[L[x] \triangleright T]\ ]\!] \xrightarrow{0, L(x, \{K\}, K_c, bnd)} \mathcal{I}[\![\ B \cup bnd, K[T]\ ]\!]$$

– $L[T]$. In this case $R = \mathbf{0}$, or $R = L[U]$. If $R = \mathbf{0}$ we have:

$$\begin{array}{l}(\textbf{new}\ h_Q\ \overline{K}(h_Q, K_a, \emptyset, B \setminus \{K\}).\mathbf{0}) + \mathcal{S}[\![\ K[L[T]], B, L[T], \{K\}\ ]\!] + \\ \mathcal{I}[\![\ B, \mathcal{A}[\![\ K[L[T]]\ ]\!]\ ]\!] \\ \quad \equiv\ (\textbf{new}\ h_Q\ \overline{K}(h_Q, K_a, \emptyset, B \setminus \{K\}).\mathbf{0}) + \\ \qquad (\textbf{new}\ h_T\ \overline{L}(h_T, K_a, \{K\}, B \setminus \{L\}).\mathcal{I}[\![\ N, K[\mathbf{0}]\ ]\!]) + \\ \qquad \mathcal{S}[\![\ K[L[T]], B, T, \{K, L\}\ ]\!] + \mathcal{I}[\![\ B, \mathcal{A}[\![\ K[L[T]]\ ]\!]\ ]\!] \\ \qquad \xrightarrow{0, \overline{L}(h_T, K_a, \{K\}, B'')} \mathcal{I}[\![\ N, K[\mathbf{0}]\ ]\!]\end{array}$$

Where h_T is the higher-order indicator for T, and B'' is $B \setminus \{L\}$ with all $b_i \in B \setminus \{L\}$ replaced by **new** b_i. Since $T = \mathbf{0}$

$$\mathcal{I}[\![\ B, K[L[T]]\ ]\!] \xrightarrow{0, \overline{L}(h_T, K_a, \{K\}, B'')} \mathcal{I}[\![\ B, K[T]\ ]\!]$$

If $R = L[U]$ we have instead, $L[T] \xrightarrow{\alpha} L[U]$. This is the case where a subkell in T is suspended, or $L[T]$ advances its execution. Let us assume that a subkell K_i in T is suspended, and T being composed of n subkells $K_1, K_2, ..., K_n$. The encoding of $L[T]$ is a process with the following structure:

$$\begin{array}{l}\textbf{new}\ h_1\ \overline{K_1}(h_1, \{K\}, B \setminus \{K_1\}).\mathcal{I}[\![\ ...\ ]\!] + \\ \textbf{new}\ h_2\ \overline{K_2}(h_2, \{K, K_1\}, B \setminus \{K_2\}).\mathcal{I}[\![\ ...\ ]\!] + \\ ... + \\ \textbf{new}\ h_n\ \overline{K_n}(h_n, \{K, K_1, K_2, ..., K_{n-1}\}, B \setminus \{K_n\}).\mathcal{I}[\![\ ...\ ]\!] + ...\end{array}$$

Where h_i is the higher-order indicator for the process inside the i-th subkell. By πSum, we have:

$$\mathcal{I}[\![\ B, K[L[T]]\ ]\!] \xrightarrow{0, \overline{K_i}(h_i, K_a, \{K, K_1, K_2, ..., K_{i-1}\}, B')} \mathcal{I}[\![\ B, K[U]\ ]\!]$$

If $L[T]$ is advancing its execution we need to show:

$$\mathcal{I}[\![\ B, \mathcal{A}[\![\ K[L[T]]\ ]\!]\ ]\!] \xrightarrow{M,\alpha'} \mathcal{I}[\![\ B, K[L[U]]\ ]\!]$$

Note $\mathcal{A}$ lifts concretions and abstractions from the inner kells towards the outside kells, eventually obtaining expressions $a(\widetilde{c}) \triangleright K[...]$, $\overline{a}(\widetilde{w})|K[...]$, and $K'[x] \triangleright K[...]$. Once the concretion or abstraction corresponding to α is outside K, the resulting process is encoded into a MMCπ expression on which, as we have shown, a corresponding MMCπ transition can be applied.

– $U|T$, with U or T having one of the forms already considered. Let us assume U is the process transitioning from U to U'. We have already shown $\mathcal{I}[\![\ B, U\ ]\!] \xrightarrow{M,\alpha'} \mathcal{I}[\![\ B, U'\ ]\!]$. Since T stays the same, we have:

$$\mathcal{I}[\![\ B, U|T\ ]\!] \stackrel{def}{=} \mathcal{I}[\![\ B, U\ ]\!]|\mathcal{I}[\![\ B, T\ ]\!] \xrightarrow{M,\alpha'} \mathcal{I}[\![\ B, U'\ ]\!]|\mathcal{I}[\![\ B, T\ ]\!]$$

Since $\mathcal{I}[\![\ B, U'\ ]\!]|\mathcal{I}[\![\ B, T\ ]\!] \equiv \mathcal{I}[\![\ B, U'|T\ ]\!]$, then

$$\mathcal{I}[\![\ B, U|T\ ]\!] \xrightarrow{M,\alpha'} \mathcal{I}[\![\ B, U'|T\ ]\!]$$

The other case is when there is a communication between U and T. In this case $\alpha' = \tau$, and R is $U'|T'$. We have already shown for L-React, L-Suspend, and L-Close:

$$\mathcal{I}[\![\ B, U|T\ ]\!] \xrightarrow{M,\tau} \mathcal{I}[\![\ B', U'|T'\ ]\!]$$

When $K[U|T]$, $\mathcal{A}$ non-deterministically advances the execution of U or T. Eventually, the interacting actions are lifted outside K, where they communicate.

Appendix C

Core API Model and Properties

See Table A.1 for special characters in the listing below and their corresponding kell-m and kμ equivalents.

```
1  process subscription(notify, filter, callback, ttl) {
2    notify(deliver, event) ->> (
3      if (@filter(event) and @ttl()) then
4        deliver(callback, event)
5      fi
6    )
7  }
8
9
10 process coreapi_debs(subscribe, unsubscribe, publish, deliver) {
11   fresh sem (
12     sem() |
13     var subsc := [] in (
14       subscribe(filter, callback, ttl, rc) ->> (
15         fresh s (
16           rc(s)
17           |
18           subscription(s, filter, callback, ttl)
19           |
20           sem() -> ((subsc :=S @cons(s, *subsc)) -> sem())
21         )
22       )
23       |
24       publish(e) ->> (
25         foreach nc in *subsc do @nc(deliver, e) done
26       )
```

```
27        |
28        unsubscribe(s) ->> (
29          sem() -> (subsc :=S @del(*subsc, s)) -> *sem()
30        )
31      )
32    )
33  }
34
35  ##############################################################################
36  # SAFETY:
37  # (1)- notification of events of interest only
38  # (2)- subscriptions precede notifications and publications
39
40  property returns_true(Rc) {
41    kEe(<Rc(T, F)>. kEe(<T()>))
42  }
43
44  property c_interested(Filter, Event) {
45    kEe(<Filter(Event, Rc)>. returns_true(Rc))
46  }
47
48  property c_active(Ttl) {
49    kEe(<Ttl(Rc)>. returns_true(Rc))
50  }
51
52  property c_of_interest_only() {
53    ~kEe(<subscribe(Filter, Callback, Ttl, Rc)>.
54      ( kEe(<deliver(Callback, Event)>)
55        &&
56        ~c_interested(Filter, Event)
57      )
58    )
59  }
60
61  property c_delivery_before_subsc() {
62    <deliver(Callback, Event)> ||
63    <-subscribe(Filter, Callback, Ttl, Rc)>.c_delivery_before_subsc()
64  }
65
66  property c_delivery_before_pub() {
67    <deliver(Callback, Event)> ||
68    <-publish(Event)>.c_delivery_before_pub()
69  }
```

```
70
71 ##############################################################################
72 # LIVENESS:
73 # (1) a published event is notified to all components interested in
74 #     the event
75
76 property c_all_notified() {
77   ( kEe(<publish(Event)>)
78     &&
79     kEe(<subscribe(Filter, Callback, Ttl, SubRc)>.
80       kEe(<SubRc(S)>.
81         c_interested(Filter, Event)
82       )
83     )
84   )
85   =>
86   ( kEe(<deliver(Callback, Event)>)
87     ||
88     kEe(<unsubscribe(S)>)
89     ||
90     kEe(<Ttl(TtlRc)>. ~returns_true(TtlRc))
91   )
92 }
93
94 ##############################################################################
95 # KELL CONTAINMENT PROPERTIES:
96
97 property c_from_site_only(Kell, Event) {
98   ~kEe(<publish(Event), Kr, ~{Kell}>)
99 }
100
101 property c_not_at_site(Site) {
102   [publish(Event), Kr , Kw].(Site in Kw).tt  &&
103   [-].c_not_at_site(Site)
104 }
105
106 property c_chained_events(E1, E2, Site) {
107   kEe(<publish(E1), Kr, >={Site}> => kE(<publish(E2)>))
108 }
109
110 ##############################################################################
111 # KELL PASSIVATION PROPERTIES:
112
```

```
113 property service_migration(Callback, Service, Event) {
114   [Callback(Event), >={Service, Machine1}].(
115     kE(<Service[X], >={Machine2}>.(Machine1 != Machine2).tt)
116   ) && [-].service_migration(Callback, Service, Event)
117 }
118
119 property forced_migration(Callback) {
120   [Callback(System)].(kE(<Service[X], ={Machine1}, ={Machine2}>))
121   && [-].forced_migration(Callback)
122 }
123
124
125 property reduce_servers(Callback) {
126   [Callback(WebServer)].(kE(<stop(WebServer)>. kE(<WebServer[X]>)))
127   && [-].reduce_servers(Callback)
128 }
129
130 process adjust_features() {
131   callback(e) -> (K[X] -> K[P])
132 }
133
134 ##############################################################################
135 # OTHER PROPERTIES:
136 # (1) unordered notification (true)
137
138 property c_unordered_notification() {
139   kEe(<publish(E1)>. kEe(<publish(E2)> .
140     kEe(<deliver(C2, E2)>. kEe(<deliver(C1, E1)>))))
141 }
```