Using a Modular Distributed Temporal Logic for In-the-large Object Specification

Juliana Küster Filipe*

Technische Universität Braunschweig, Informatik, Abt. Datenbanken
Postfach 3329, D–38023 Braunschweig, Germany
e–mail: J.Kuester-Filipe@tu-bs.de

Abstract

Our general goal is to provide a semantic foundation for in-the-large specification of distributed information systems. We use TROLL, an object-oriented formal language, for system specification. Our claim is that objects are not enough as a modularisation unit when it comes to deal with very large systems. An intermediary concept between the system and the objects is needed for allowing reusability of specifications and provide a clearer system structure. Enriching TROLL with a module concept enforces us to develop new theoretical constructs ensuring an appropriate underpinning of the language. A modular distributed temporal logic MDTL has been developed which describes the dynamic aspects of modular systems. The main features of the logic are its own modularity, the ability to express inter-module (a)synchronous communication and intra-module concurrency. It also seems promising to support module refinement. MDTL is based on n-agent logics and interpreted over labelled prime event structures. In this paper we present the logic and semantics underlying the modular object-oriented specification language by means of a small toy example.

1 Introduction

The object-oriented paradigm has increased its popularity over the past years. This essentially due to its advantages in software development, maintenance and reusability. Object-orientation also seems to be widely accepted at the level of software specification and design. Moreover, we defend the use of a formal object-oriented approach to system specification. Being precise and unambiguous, formal approaches have a rigorous semantics and allow system validation, correctness verifiability and forecasting of system properties.

The formal object-oriented language TROLL [11] has been developed for the conceptual modelling of distributed information systems. Information systems are reactive systems with one or more databases and application programs. In TROLL, a system is regarded as a collection of communicating objects. Calculations are performed concurrently by the objects involved, interacting when needed. Objects are encapsulated units of structure and behaviour with explicitly defined interfaces through which they can be accessed. Objects have a unique and state independent identity. Furthermore they have an internal state and actions (methods) that determine their behaviour. Objects have a sequential behaviour. The internal state is given by

*Work reported here was supported by the DFG under Eh75/11-1 and partially by the EU under ESPRIT-IV WG 22704 ASPIRE.

the current values of the attributes, enabled and occurring actions. Object classes allow the description of similar objects. Interaction among objects is done by calling actions of other objects to happen synchronously. Objects may be specialised or composed into complex objects through aggregation.

In Troll, as well as in most object-oriented languages, the only unit of modularisation is the object class. However, with the increasing complexity of software systems and the need of having parameterised units for reuse of specifications, the object is unsuitable for a modularisation unit [8, 24, 23]. Objects are too small to be effectively reused or allow a good system structure [25, 21]. Finally, experience using Troll in a joint industrial project with the PTB (German Federal Institute of Weights and Measures) developing a large information system [12] has shown that reusability is too hard to achieve this way, and a clearer system structure would improve the development process.

These reasons made us recently work towards a module concept for Troll. First considerations on the language extension have recently been reported in [5]. Both means of communication are to be supported by the modular language Troll since we believe that, at a high level of abstraction, synchronous and asynchronous communication mechanisms should be used whenever they suit the application the most. E.g., a mailing system is a typical example for asynchronous communication while the telephone system is an example for synchronous one.

In this paper we focus on the theoretical foundations of a module concept for object-oriented languages. It had the theory underlying non-modular Troll as a starting point. Inherent to the extension of the language by a new structuring concept, various new aspects have to be considered. These include module encapsulation, intra-module concurrency, module composition, communication, parameterisation and refinement.

Object systems specified in non-modular Troll [11] can be translated to formulae of Dtl [6] which is an extension of linear temporal logic based on n-agent logic [17]. The logic is interpreted in labelled prime event structures. Labelled prime event structures [20] constitute a non-interleaving model of concurrency appropriate for denotational semantics (see [22]) and describing the behaviour of object systems as needed.

In Dtl, each object has a local logic allowing it to make assertions about itself and system properties from its local viewpoint. Such system properties can be expressed based on the inter-object communication. Dtl as well as n-agent logics [17, 19, 4] are based on the sequentiality of objects and agents. Dtl and the n-agent logic in [17] do not talk about concurrency and only focus on concepts like object/agent locality and communication. Moreover, Dtl only deals with synchronous object communication. Most of the developed logics for dealing with distributed systems are based on synchronous communication [1, 2]. We have developed a modular logic Mdtl as an extension of Dtl supporting modules and both synchronous and asynchronous communication [16]. Modules are in general not sequential and therefore a concurrency operator was introduced in the style of [19, 4]. Mdtl is interpreted over labelled prime event structures. Module model constructions for (a)synchronous concurrent composition, encapsulation (or abstraction), renaming and parameter actualisation have been given in [15, 13, 14]. The model theory together with Mdtl allows a whole range of issues, like reasoning and verification of modular systems.

In this paper we introduce a small example focusing on inter-module communication, and present the logic and semantics by means of this example.

This paper is structured in the following way. In the next section we present informally the idea of the module concept for Troll. We describe the modular structure of a system from a theoretical viewpoint. In section 3 the logic Mdtl and its semantics are given and explained by means of a simple example. This paper finishes mentioning related work and discussing results and further work.

2 Object-Oriented Modules

A system may be seen as a collection of distributed and interacting object modules. A module can be more than an object and denote a parameterisable system part with intra-module concurrency and interfaces through which it can communicate with other modules. An object module is either compound, in which case it consists of smaller modules, or basic. The basic form of a module consists of one or more objects. Figure 1 illustrates the modular structure of a system.

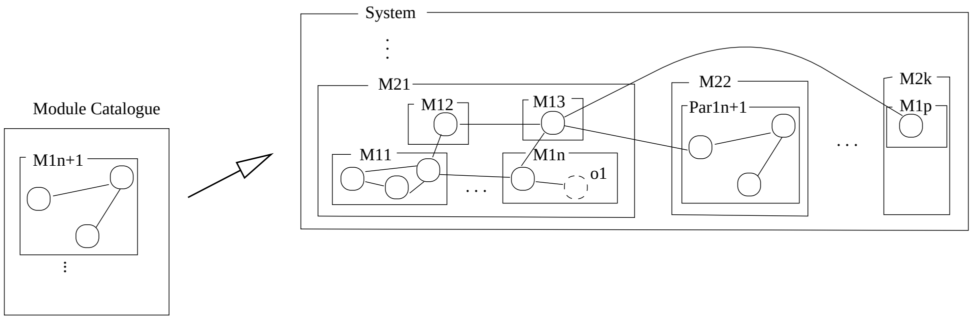

Figure 1: Structuring systems through different module levels.

Circles represent objects, and lines between circles denote object interaction. Frames represent modules with their name written on top, e.g., `Mlk` where `l` indicates the module level and `k` its number within the level. The number of module levels and modules within the same level are finite. Moving bottom up, objects are grouped into modules, for instance `M11,..., M1p` on the first level. Combining modules of the same level and adding interactions between them, we obtain modules in the subsequent level. Module interaction is done by the objects belonging to them. The last module level corresponds to the system itself.

A module may declare several views of itself for *export*, e.g., `M1n` may declare a view hiding object `o1`. A module *imports* a view of modules from the previous level, e.g., `M21` imports what `M1n` declares as visible. In what follows, a module from a certain level is seen as the collection of all the visible objects of the imported modules of the previous level. Module bounds from previous levels are forgotten and just their visible objects (interface objects) are relevant. A module catalogue contains modules for reuse. A module can have a *parameter* part which is again a module, e.g., `M22` contains a parameter `Par1n+1`. A parameter module is treated as any other module from its level. Parameter modules can be instantiated by modules from the catalogue. Modules from the same level may interact with each other. Communication is considered to be synchronous or asynchronous. Synchronous communication corresponds to the simultaneous occurrence of actions of the objects involved in the communication. Since a novelty of MDTL concerns asynchronous communication, we will focus on this means of communication. We present informally the conditions we impose on the asynchronous communication.

We consider an asynchronous mechanism where the send of a message is non-blocking, meaning that an object can deliver a message without waiting for it to reach its destination. For asynchronous communication among modules what we need is a calling action from one module object, say a *send* action, and one or more distinct modules, where each module has one object with a corresponding called action, say a *receive* action. If we have only one-to-one communication we will have one calling module object with an action *send* and one called module with a unique object and corresponding action *receive*. In case of multicasting (or broadcasting) we will have one calling module object with an action *send* and some (or all) modules, where each has a unique receive object and corresponding action *receive*.

Assumption 1 There is a unique correspondence between the occurrence of a *send* and a *receive* actions between two module objects.

We distinguish among communication and non-communication actions. Furthermore, a communication action may either be a *send* or a *receive* action but never both.

Assumption 2 An action may be a communication or a non-communication action. A communication action is either a *send* or a *receive* action.

We also assume in our approach that asynchronous communication is safe, i.e., for each *send* messages there exists a *receive* action in the called module(s).

Assumption 3 Asynchronous communication is safe.

The next assumption concerns the order of receipt of sent messages.

Assumption 4 The overtaking of distinct messages is possible, i.e., the order of the occurrences of the corresponding *receive* actions need not be preserved. The order is considered to be preserved if the same message is sent more than once.

A further assumption on asynchronous communication we take for our specification language, is that cycles are not allowed, i.e., $M_1.a_1$ asynchronously calls $M_2.a_2$ etcetera till $M_n.a_n$ asynchronously calls $M_1.a_1$. Furthermore, by assumption 2, an action cannot be simultaneously a *send* and a *receive* action, but a *receive* action can call synchronously a *send* action to happen.

Assumption 5 The specification language does not allow calling cycles.

From assumptions 2 and 5 we understand that asynchronous communication is antisymmetric. From assumption 2 we know that we cannot express:

$$M_1.a_1 \ asyn\ calls\ M_2.a_2 \ asyn\ calls\ M_1.a_1$$

as a_1 is either a *send* or a *receive* action. On the other side, assumption 5 makes sure that the following is not possible:

$$M_1.s_1 \ asyn\ calls\ M_2.r_2 \ sync\ calls\ M_2.s_2 \ asyn\ calls\ M_1.r_1 \ sync\ calls\ M_1.s_1$$

3 MDTL

In the sequel we assume as given an extended data signature $\Sigma = (S, \Omega)$ where S is a set of sorts (data and object sorts, i.e., $S = S_D \cup S_O$) and Ω an $S^* \times S$-indexed family of sets of operation symbols. Let $X = \{X_s\}_{s \in S}$ be an S-indexed family of sets of variables. The interpretation structures for extended data signatures are Σ-Algebras. For more details consult [7].

At a certain module level l, we have a finite set of modules given by $Mod^l = \{1, ..., n\}$. Let $m \in Mod^l$. A *module signature* for m is given by $\Sigma_m = (Id_m, At_m, Ac_m)$ where Id_m is a set of *object identifiers* (the module object instances), At_m is an $Id_m \times S$-indexed family of sets $At_m = \{At_{m_{i,s}}\}_{i \in Id, s \in S}$ of *attribute symbols*, and Ac_m is an $Id_m \times S^*$-indexed family of sets $Ac_m = \{Ac_{m_{i\omega}}\}_{i \in Id\ \omega \in S^*}$ of *action symbols*. For an arbitrary $\alpha \in At_{m_{i,s}}$, α is an attribute of object i and type s. For an arbitrary $\beta \in Ac_{m_{is_1 \ldots s_n}}$, β is an action of object i with n parameters of sorts $s_1 \ldots s_n$. Accordingly to the assumption 2 of the previous section, actions are subdivided into asynchronous communication actions and remaining actions, i.e., $Ac_m = Ac_m^{ac} \uplus Ac_m^{\neg ac}$. Moreover, asynchronous communication actions may either be *send* or *receive* actions, i.e., $Ac_m^{ac} = SAc_m \uplus RAc_m$ whereby SAc_m and RAc_m are indexed families as explained next. A send action has to know the modules to which it is sending a message, the corresponding object within the modules which is going to receive it, as well as its receive action. A receive action knows from which module and module object the message was sent and what the sending action was. SAc_m and RAc_m are defined as follows:

- $SAc_{m_{i\omega}}^{Y_1 \ldots Y_n}$, where $n \geq 1$ and for $1 \leq p \leq n$, $Y_p = \langle k, j, \beta \rangle$ with $k \in Mod$, $k \neq m$, $j \in Id_k$ and $\beta \in RAc_{k_{j\omega}}^{\langle m,i,- \rangle}$, and
- $RAc_{m_{i\omega}}^{Y}$, where $Y = \langle k, j, \beta \rangle$ with $k \in Mod$, $k \neq m$, $j \in Id_k$ and $\beta \in SAc_{k_{j\omega}}^{\langle m,i,- \rangle \ldots}$.

For an arbitrary action α with parameter x of sort s, $\alpha(x) \in SAc_{m_{is}}^{\langle k,j,\beta \rangle \ldots}$ means that object i of module m is sending a message to one or more modules including module k, more concretely to its object j and such that $\beta(x)$ is the corresponding receive action at j. Moreover, $\beta(x) \in RAc_{k_{js}}^{\langle m,i,\alpha \rangle}$. In the specification language objects with similar structure and behaviour are grouped in *object classes*, i.e., attributes and actions are defined at the class level and objects are declared as instances of the class. We will use the notation $\alpha \equiv i.a$ for attributes or actions where i stands for the object identifier and a for the attribute or action name as declared in the corresponding object class. The actions in the next example, even though they do not provide this intuition, are considered here as being atomic in duration.

Example 3.1 Consider a small and simplified extract of a music world. At a certain module level there is a compound module `MUSIC_SCHOOL`$\otimes$`BOOKSHOP` importing views of two modules `MUSIC_SCHOOL` and `BOOKSHOP` and adding further interactions among them.

Within the module `MUSIC_SCHOOL` we consider declared the object classes `Cello_Students` and `Piano_Students`. The class `Cello_Students` has an attribute `repertoire` of type set of string, and the actions `request, play_solo, play_duet` and `receive`, each of them with an argument of type string. For the class `Piano_Students` just consider the actions `play, sing` and `dance` with an argument of type string. The class `Cello_Students` has an object instance `Jane` and the class `Piano_Students` an object instance `Mary`. The intra-module interaction is established between `Jane` and `Mary` whenever `Jane` wants to play the duet piece `m` accompanied by the piano, i.e., the occurrence of the action `Jane.play_duet(m)` implies the simultaneous occurrence of the action `Mary.play(m)` (synchronisation). The module `MUSIC_SCHOOL` does not make visibility restrictions to importing modules.

The module `BOOKSHOP` contains two object classes `Bookshop_Employee` and `Publishers`, with object instances `Bookseller` and `IntMusicComp` respectively. For the class `Bookshop_Employee` consider an attribute `store` of type set of string, and actions `receive, deliver, request_pub` and `receive_pub` each of them with an argument of type string. The class `Publishers` only has two actions `receive` and `deliver` both with an argument of type string. Intra-module communication is considered to be asynchronous. Whenever the `Bookseller` receives a request which he cannot satisfy (the requested book is not available in the store), he mails the `IntMusicComp` for

the requested book. After receiving it from the publisher, the `Bookseller` may deliver it. The module `BOOKSHOP` only offers for export a view with object `Bookseller` visible (`IntMusicComp` is hidden).
The inter-module interaction is established between `Jane` and the `Bookseller` and is of asynchronous nature. `Jane` can only play the music pieces of her `repertoire`. To enrich her `repertoire` she has to acquire new music books. She mails the `Bookseller` a `request(m)` while she can go on playing her old pieces. As the communication is assumed to be safe, sometime later the `Bookseller` will `receive(m)` the request. When the requested music book `m` is available in the `store` the `Bookseller` will `deliver(m)` it by mail to `Jane`. We use abbreviations for module names. Formally we have the following view signatures:

$Mod^l = \{\mathtt{M\_S}, \mathtt{B}\}$ and

$\Sigma_{\mathtt{M\_S}} = (Id_{\mathtt{M\_S}}, At_{\mathtt{M\_S}}, Ac_{\mathtt{M\_S}})$ with

$$
\begin{array}{l}
Id_{\mathtt{M\_S}} = \{\mathtt{Jane}, \mathtt{Mary}\}\\
At_{\mathtt{M\_S}}:\\
\quad At_{\mathtt{M\_S}_{\mathtt{Jane}, set(string)}} = \{\mathtt{Jane.repertoire}\}\\
Ac_{\mathtt{M\_S}}:\\
\quad Ac_{\mathtt{M\_S}_{\mathtt{Jane}, string}} = \{\mathtt{Jane.request}, \mathtt{Jane.play\_solo}, \mathtt{Jane.play\_duet},\\
\qquad \mathtt{Jane.receive}\}\\
\qquad Ac^{\neg ac}_{\mathtt{M\_S}_{\mathtt{Jane}, string}} = \{\mathtt{Jane.play\_solo}, \mathtt{Jane.play\_duet}\}\\
\qquad Ac^{ac}_{\mathtt{M\_S}_{\mathtt{Jane}, string}} = \{\mathtt{Jane.request}, \mathtt{Jane.receive}\}\\
\qquad\quad SAc^{\langle \mathtt{B}, \mathtt{Bookseller}, \mathtt{Bookseller.receive}\rangle}_{\mathtt{M\_S}_{\mathtt{Jane}, string}} = \{\mathtt{Jane.request}\}\\
\qquad\quad RAc^{\langle \mathtt{B}, \mathtt{Bookseller}, \mathtt{Bookseller.deliver}\rangle}_{\mathtt{M\_S}_{\mathtt{Jane}, string}} = \{\mathtt{Jane.receive}\}\\
\quad Ac_{\mathtt{M\_S}_{\mathtt{Mary}, string}} = \{\mathtt{Mary.play}, \mathtt{Mary.sing}, \mathtt{Mary.dance}\}\\
\qquad Ac^{\neg ac}_{\mathtt{M\_S}_{\mathtt{Mary}, string}} = \{\mathtt{Mary.play}, \mathtt{Mary.sing}, \mathtt{Mary.dance}\}
\end{array}
$$

$\Sigma_{\mathtt{B}} = (Id_{\mathtt{B}}, At_{\mathtt{B}}, Ac_{\mathtt{B}})$ with

$$
\begin{array}{l}
Id_{\mathtt{B}} = \{\mathtt{Bookseller}\}\\
Ac_{\mathtt{B}}:\\
\quad Ac_{\mathtt{B}_{\mathtt{Bookseller}, string}} = \{\mathtt{Bookseller.receive}, \mathtt{Bookseller.deliver}\}\\
\qquad \mathtt{Bookseller.request\_pub}, \mathtt{Bookseller.receive\_pub}\}\\
\qquad Ac^{\neg ac}_{\mathtt{B}_{\mathtt{Bookseller}, string}} = \{\mathtt{Bookseller.request\_pub}, \mathtt{Bookseller.receive\_pub}\}\\
\qquad Ac^{ac}_{\mathtt{B}_{\mathtt{Bookseller}, string}} = \{\mathtt{Bookseller.receive}, \mathtt{Bookseller.deliver}\}\\
\qquad\quad SAc^{\langle \mathtt{M\_S}, \mathtt{Jane}, \mathtt{Jane.receive}\rangle}_{\mathtt{B}_{\mathtt{Bookseller}, string}} = \{\mathtt{Bookseller.deliver}\}\\
\qquad\quad RAc^{\langle \mathtt{M\_S}, \mathtt{Jane}, \mathtt{Jane.request}\rangle}_{\mathtt{B}_{\mathtt{Bookseller}, string}} = \{\mathtt{Bookseller.receive}\}
\end{array}
$$

□

MDTL consists of local module logics at a given module level l. Each module $m \in Mod^l$ has a local logic MDTL^l_m. The local logic MDTL^l_m allows to make assertions about the module m itself and through communication about other modules. MDTL may be defined as follows:

$$
\begin{array}{ll}
\mathrm{MDTL}^l ::= \{\mathrm{MDTL}^l_m\}_{m \in Mod^l} & \\
\mathrm{MDTL}^l_m ::= m.H^l_m \mid m.C^l_m & \\
H^l_m ::= \mathrm{ATOM}^l_m \mid \neg(H^l_m) \mid (H^l_m \Rightarrow H^l_m) \mid (H^l_m\ \mathcal{U}\ H^l_m) \mid (H^l_m\ \mathcal{S}\ H^l_m) \mid \Delta(H^l_m) \mid \forall_x(H^l_m) & \\
C^l_m ::= S^l_m \Rightarrow k.S^l_k \mid AS^l_m \Rightarrow F_k\ AR^l_k \mid AR^l_m \Rightarrow P_k\ AS^l_k & \text{for some } k \in Mod^l, m \neq k\\
S^l_m ::= \odot AC^{\neg ac}_{\Sigma_m}(X) \mid S^l_m \wedge P^l_m \mid \forall_x S^l_m \mid \exists_x S^l_m & \\
AS^l_m ::= \odot SAC_{\Sigma_m}(X) \mid AS^l_m \wedge P^l_m \mid \forall_x AS^l_m \mid \exists_x AS^l_m & \\
AR^l_m ::= \odot RAC_{\Sigma_m}(X) \mid AR^l_m \wedge P^l_m \mid \forall_x AR^l_m \mid \exists_x AR^l_m &
\end{array}
$$

$$P^l_m ::= \text{ATOM}^l_m \mid P^l_m \wedge P^l_m \mid \forall_x P^l_m \mid \exists_x P^l_m$$

$$\text{ATOM}^l_m ::= \mathit{true} \mid T_\Sigma(X)\ \theta\ T_\Sigma(X) \mid AT_{\Sigma_m} \theta\ T_\Sigma(X) \mid \rhd AC_{\Sigma_m}(X) \mid \odot AC_{\Sigma_m}(X)$$

The local logic of module m is split into a module *home* logic H^l_m and a module *communication* logic C^l_m. Both of them are defined over another logic, namely the *state* logic P^l_m.

P^l_m is a restricted first order predicate logic. P^l_m is used to express states of a module and its objects and defines labels for our models. A state is given by the current values of the attributes, enabled and occurring actions. A state formula is written as a conjunction of such aspects. $T_\Sigma(X)$, AT_{Σ_m} and $AC_{\Sigma_m}(X)$ represent data, attribute and action terms respectively. $AC^{\neg ac}_{\Sigma_m}(X), SAC_{\Sigma_m}(X)$ and $RAC_{\Sigma_m}(X)$ denote special action terms, namely non-asynchronous communication, send and receive action terms respectively. An atomic formula of P^l_m can be the boolean constant *true*; the predicate θ applied to two data terms or to an attribute and a data term, where θ is an comparison predicate (e.g., $=, \leq, \ldots$); the predicate $\rhd$ (enabling) applied to an action term; or the predicate $\odot$ (occurrence) applied to an action term. Notice that the predicates $\rhd$ and $\odot$ are needed to be able to distinguish among actions that may occur next (are enabled) and are occurring. A state formula is a conjunction of atomic formulae with quantified variables.

H^l_m is a first order predicate temporal logic such that $P^l_m \subset H^l_m$. Atomic formulae in H^l_m are the atomic formulae in P^l_m. Formulae in H^l_m can be obtained by applying successively the connectives $\neg$ and $\Rightarrow$, the temporal operators $\mathcal{U}$ (until) and $\mathcal{S}$ (since), the operator Δ and the $\forall$ quantifier to atomic formulae. The well-known connectives of propositional calculus such as $\wedge$, $\vee$ and $\Leftrightarrow$ are defined in terms of $\neg$ and $\Rightarrow$ in the usual way. The other temporal operators like *next* X, *sometime in the future* F, *always in the future* G and *sometime in the past* P can be derived from $\mathcal{U}$ and $\mathcal{S}$ as well (cf. [6] for details). The new operator Δ is a concurrency operator whose intention will become clear later on.

The communication logic C^l_m allows to express communication among several objects from distinct modules of level l. Notice that intra-module communication is expressed as a formula in the home logic. A formula in the logic C^l_m exhibits knowledge about other modules through communication and from the local viewpoint of module m. There are three possible statements in the logic concerning communication, the first refers to synchronous while the latter refer to asynchronous communication. The synchronisation formulae S^l_m denote P^l_m formulae containing at least one occurrence of a non-asynchronous communication action within module m. Moreover, $S^l_m \Rightarrow k.S^l_k$ allows to express synchronous callings of actions of objects from distinct modules. Similarly, AS^l_m and AR^l_m denote P^l_m formulae containing occurrences of send and receive communication actions respectively. A state logic formula can for instance be $\phi \equiv \odot i.a \wedge \rhd i.b$. Depending on the kind of action $i.a$ denotes ϕ might be a formula in S^l_m, AS^l_m or AR^l_m. In the communication logic we use the Mod-indexed temporal operators F and P only to increase readability as they allow to express the asynchronous communication mechanism more directly. Non-indexed temporal operators refer to module locality while the Mod-indexed temporal operators refer to communication.

In what follows, to simplify the presentation, we omit module identifiers in the formulae, i.e., instead of $m.\varphi$ for a formula in MDTL_m we just write φ.

Example 3.2 In the following we present some examples of formulae for the module `MUSIC_SCHOOL` in its local logic $\text{MDTL}^l_{\texttt{M\_S}}$. φ_1 and φ_2 are formulae in the home logic $H^l_{\texttt{M\_S}}$, while φ_3 and φ_4 are formulae in the communication logic $C^l_{\texttt{M\_S}}$.

The next formula indicates the synchronous interaction between `Jane` and `Mary`.

$$\varphi_1 \equiv \forall_m(\odot\texttt{Jane.play\_duet(m)} \Rightarrow \odot\texttt{Mary.play(m)})$$

The next formula states that meanwhile `Jane` receives a music book, `Mary` sings some song (concurrently).

$$\varphi_2 \equiv \forall_{\mathtt{m_1}} \odot \texttt{Jane.receive(m}_1\texttt{)} \Rightarrow \Delta\ \exists_{\mathtt{m_2}} \odot \texttt{Mary.sing(m}_2\texttt{)}$$

The next two formulae reflect the interaction with module `BOOKSHOP` from the local viewpoint of the module `MUSIC_SCHOOL`. Whenever `Jane` requests a music book at the `BOOKSHOP`, the `Bookseller` will sometime later receive the request (asynchronous communication is safe).

$$\varphi_3 \equiv \forall_m \odot \texttt{Jane.request(m)} \Rightarrow F_{\mathtt{B}} \odot \texttt{Bookseller.receive(m)}$$

Whenever `Jane` receives a music book from the `BOOKSHOP`, she knows that sometime in the past the `Bookseller` delivered it.

$$\varphi_4 \equiv \forall_m \odot \texttt{Jane.receive(m)} \Rightarrow P_{\mathtt{B}} \odot \texttt{Bookseller.deliver(m)}$$

Conversely, in the `BOOKSHOP`'s communication logic $C^l_{\mathtt{B}}$ following statement can be made:

$$\phi_1 \equiv \forall_m(\odot\texttt{Bookseller.receive(m)} \Rightarrow P_{\mathtt{M\_S}} \odot \texttt{Jane.request(m)})$$

Examples of formulae in the `BOOKSHOP`'s home logic $H^l_{\mathtt{B}}$ are given next. The next formula states that the `Bookseller` can only deliver a book if it is available in the store. It uses a data operation *isIn* which is predefined in TROLL.

$$\phi_2 \equiv \forall_m(\rhd\texttt{Bookseller.deliver(m)} \Rightarrow \texttt{m}\ isIn\ \texttt{Bookseller.store})$$

If the `Bookseller` receives a request of a book which is not available, he will next send a corresponding request for it (to whom is not visible at this level).

$$\phi_3 \equiv \forall_m(\odot\texttt{Bookseller.receive(m)} \Rightarrow \neg(\texttt{m}\ isIn\ \texttt{Bookseller.store}) \Rightarrow X \odot \texttt{Bookseller.request\_pub(m)})$$

□

The logic is interpreted over labelled prime event structures. Prime event structures allow a description of distributed computations as event occurrences together with a causality and a conflict relations between them. The causal relation implies a (partial) order among event occurrences, and the conflict denotes a choice. Events in conflict cannot belong to the same system, module or object run. A run is often called life cycle. Formally, a prime event structure is a triple $E = (Ev, \rightarrow^*, \#)$, where Ev is a set of events, $\rightarrow^*$ and $\#$ are binary relations on events called causality (a partial order) and conflict (irreflexive and symmetric) respectively. $e_1 \rightarrow^* e_2$ means that whenever e_2 occurs e_1 must have occurred sometime before. However, e_2 does not have to occur. We shall use the notation $e_1 \rightarrow e_2$ for immediate causality (no other event in between), and $e_1 \rightarrow^+ e_2$ whenever $e_1 \rightarrow^* e_2$ and $e_1 \neq e_2$. $e_2 \# e_3$ means that these events are in conflict, i.e., either one or the other may occur but never both. Conflict propagates over causality, i.e., if $e_1 \# e_2 \rightarrow^* e_3$ then $e_1 \# e_3$. Two events which are neither in conflict nor related by causality are said to be concurrent. Concurrent events are written $e_1\ co\ e_4$. An important concept for defining a run is a configuration. A configuration C is a subset of events $C \subseteq Ev$ satisfying both (1) C is conflict free, and (2) C is downwards closed. C is conflict free iff it does not contain events in conflict, i.e., $\forall_{e_1,e_2 \in C} \neg(e_1 \# e_2)$. C is downwards closed iff for any event in C all its predecessors are in C as well, i.e., $\forall_{e_1 \in C}$ if $e_0 \rightarrow^* e_1$ then $e_0 \in C$. A life cycle (or run) is a maximal configuration, i.e., no more events can be added without violating one of the conditions. Notice that a configuration or life cycle may contain events in concurrency. These concepts will be illustrated in the next example. For a detailed presentation on event structures see, e.g., [20].

In order to be able to use event structures as a model for our object modules, we have to attach to them some more information in the form of event labels. We introduce a labelling function μ as a total function attaching to each event a state formula. A prime event structure enriched by a labelling function is called a labelled prime event structure. Each module of a given level l has a model $(E_m, \mu_m : Ev_m \rightarrow P^l_m)$.

Example 3.3 An extract of a model for module `MUSIC_SCHOOL` is given in figure 2. Events are represented by circles, the arrows indicate (immediate) causality, while conflicting events are separated by #. Events not related by causality nor by conflict are concurrent (e.g., e_5 *co* e_6). A module model is obtained by concurrent (synchronous or asynchronous) composition of the

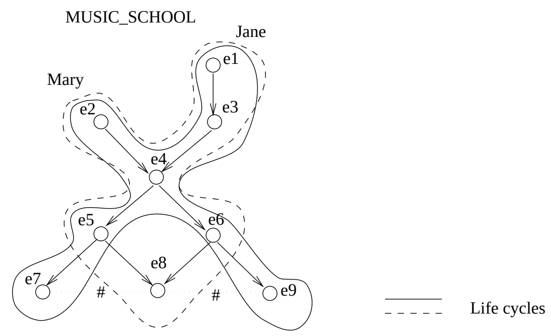

Figure 2: An extract of a model for `MUSIC_SCHOOL` showing interaction among `Jane` and `Mary`.

object models belonging to the module. A model for `MUSIC_SCHOOL` has all the events belonging only to `Mary`, those belonging only to `Jane` and shared events. Shared events denote synchronisation (e.g., events e_4 and e_8). The figure shows two possible module life cycles.

Consider the labelling function $\mu_{\texttt{M\_S}} : Ev_{\texttt{M\_S}} \rightarrow P^l_{\texttt{M\_S}}$ with some of the labels as follows:

$e_2 \longmapsto$ $\odot$`Mary.dance("Salsa")` $\wedge \forall_{\texttt{m}_1,\texttt{m}_2,\texttt{m}_3}$(`▷Mary.play(m1)` $\wedge$ `▷Mary.dance(m2)`$\wedge$
`▷Mary.sing(m3))`

$e_3 \longmapsto$ $\odot$`Jane.request("ChopinOpus3")` $\wedge$ `▷Jane.play_solo("BachSuites")`$\wedge$
`▷Jane.play_duet("BrahmsOpus38")` $\wedge$ `Jane.repertoire` $=$ {`"BachSuites"`, `"BrahmsOpus38"`}

$e_4 \longmapsto$ $\odot$`Mary.play("BrahmsOpus38")` $\wedge$ $\odot$`Jane.play_duet("BrahmsOpus38")` $\wedge \ldots$

$e_5 \longmapsto$ $\odot$`Mary.sing("MadreDeusOMar")` $\wedge \ldots$

$e_6 \longmapsto$ $\odot$`Jane.receive("ChopinOpus3")` $\wedge$ `Jane.repertoire` $=$ {`"BachSuites"`, `"BrahmsOpus38"`, `"ChopinOpus3"`} $\wedge \ldots$

$e_8 \longmapsto$ $\odot$`Mary.play("ChopinOpus3")` $\wedge$ $\odot$`Jane.play_duet("ChopinOpus3")` $\wedge \ldots$

A model for the module `BOOKSHOP` is obtained by asynchronous concurrent composition of models of its objects `Bookseller` and `IntMusicComp`. Since the module hides the object `IntMusicComp` in its export declaration, the model has to be restricted. We just provide the intuition behind the model construction for hiding. More details can be found in [13]. Consider figure 3 where an extract of a model for `BOOKSHOP` before (left) and after (right) export is given.

All the local events of `IntMusicComp` are removed from the exported model part. Since the communication is asynchronous no shared events from `IntMusicComp` and `Bookseller` exist.

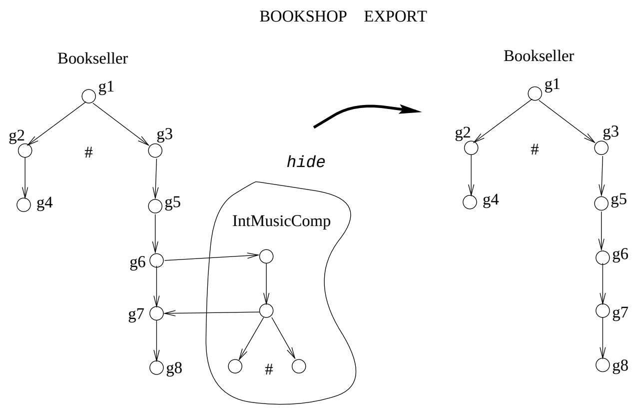

Figure 3: An extract of a model for BOOKSHOP before and after export.

Shared events would have been kept with the labels reflecting only attributes or actions from the visible objects. Consider the labelling function $\mu_{\mathtt{B}} : Ev_{\mathtt{B}} \to P^l_{\mathtt{B}}$ with some of the labels as follows:

$g_2 \longmapsto \odot\mathtt{Bookseller.receive("ChopinOpus3")} \wedge \mathtt{Bookseller.store} = \{\mathtt{"ChopinOpus3",...}\} \wedge \ldots$

$g_4 \longmapsto \odot\mathtt{Bookseller.deliver("ChopinOpus3")} \wedge \ldots$

$g_5 \longmapsto \odot\mathtt{Bookseller.receive("ChopinOpus3")} \wedge \mathtt{Bookseller.store} = \{\mathtt{"BachSuites"}\} \wedge \ldots$

$g_6 \longmapsto \odot\mathtt{Bookseller.request\_pub("ChopinOpus3")} \wedge \ldots$

$g_7 \longmapsto \odot\mathtt{Bookseller.receive\_pub("ChopinOpus3")} \wedge \mathtt{Bookseller.store} = \{\mathtt{"BachSuites"},$
$\mathtt{"ChopinOpus3"}\} \wedge \ldots$

$g_8 \longmapsto \odot\mathtt{Bookseller.deliver("ChopinOpus3")} \wedge \ldots$

□

A model for level l is (E^l, μ^l) and can be obtained by concurrent composition of the module models of the level. A model construction for (a)synchronous concurrent composition has been developed in [15]. Intuitively (E^l, μ^l) can be understood as a concurrent union of the module models glued at shared events and where the relations of the module models are kept. An event in such a model is either a *local* event, i.e., belonging just to one module model, or a *shared* event, i.e., belonging to more than one module models. A shared event reflects action synchronisation of objects from different modules and it is labelled by the conjunction of labels of the corresponding module models. A local event label is the corresponding label in the module model. If there is a causality relation among events from different module models it indicates asynchronous communication among the modules.

Example 3.4 Figure 4 presents an extract of a model for MUSIC_SCHOOL⊗BOOKSHOP obtained by asynchronous concurrent composition of the corresponding views of modules MUSIC_SCHOOL and BOOKSHOP. Conflict propagation is not explicitly indicated in the model.

If Jane sends a request at event e_3 to the BOOKSHOP she is not possibly aware of the local life cycle the Bookseller is carrying out, i.e., the request has to reach the Bookseller in any of his possible local life cycles. Therefore, and as depicted in the model of figure 4, event e_3 is linked to event g_2 and g_5.

Similarly, the receipt of a previously requested music book for Jane should also be independent of the life cycle chosen by the Bookseller. However, taking event e_6 from the model of figure 2

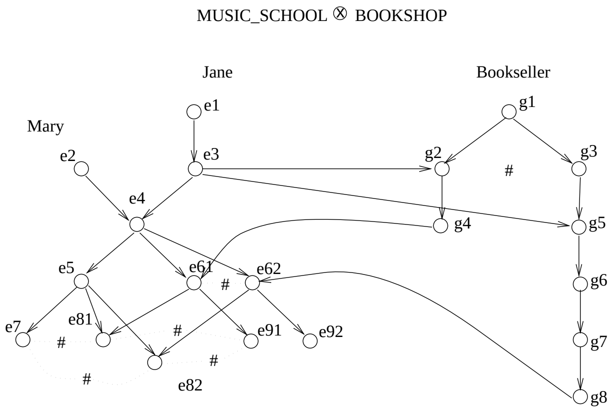

Figure 4: An extract of a model showing interaction among the modules `MUSIC_SCHOOL` and `BOOKSHOP`.

as an immediate successor of events g_4 and g_8 we do not obtain a labelled prime event structure anymore. This is due to the property of conflict propagation and it would make e_6 a self-conflicting event (not allowed, as conflict is irreflexive). Therefore, while constructing a valid model for the (a)synchronous concurrent composition of modules, a multiplication of events is necessary. In the example, event e_6 had to be duplicated originating two events e_{61} and e_{62} with the same label. Naturally, all the future events had to be multiplied as well. For more details on this characteristic of prime event structures or on the construction itself we refer the interested reader to [3, 26, 15] among others. □

In what follows assume a valuation v over the variables X defined over the interpretation of the extended data signature (i.e., $v = \{v_s : X_s \to [\![s]\!]\}$). We shall omit further details as they are not essential for understanding MDTL.
We now define the semantics of the logic MDTL. Let $M = (E^l, \mu^l)$ be a model for level l and obtained by composition of the models of the modules in Mod^l. Let v be a valuation over the variables, $m, k \in Mod^l$, $e \in Ev^l_m$, $\sigma \in P^l_m$, $m.\varphi, m.\psi$ be formulae in MDTL$_m$ and ϕ a formula in C^l_k. Satisfaction of a formula $m.\varphi$ for a model M at an event e with valuation v is denoted $M, e, v \models_m m.\varphi$. To increase readability, we omit the $m.$ notation for formulae in MDTL$_m$ while defining satisfaction $\models_m$. Satisfaction $\models_m$ is defined inductively as follows:

1. $M, e, v \models_m \sigma$ holds iff $\mu^l(e) \models_{P^l_m} \sigma$,
2. $M, e, v \models_m \neg\varphi$ holds iff $M, e, v \models_m \varphi$ does not hold,
3. $M, e, v \models_m (\varphi \Rightarrow \psi)$ holds iff $M, e, v \models_m \varphi$ implies $M, e, v \models_m \psi$,
4. $M, e, v \models_m \forall_{x:s}\varphi$ holds iff $M, e, v[x \mapsto a] \models_m \varphi$ holds for each $a \in [\![s]\!]$,
5. $M, e, v \models_m (\varphi\, \mathcal{U}\, \psi)$ holds iff for each life cycle in E^l_m containing e there is some event $e' \in Ev^l_m$, with $e \to^+ e'$ such that $M, e', v \models_m \psi$ holds, and there is a finite chain $\{e_1, \ldots, e_n\} \subseteq Ev_m$ such that $e \to e_1 \to \ldots \to e_n \to e'$ where $M, e_p, v \models_m \varphi$ holds for each e_p with $1 \leq p \leq n$,
6. $M, e, v \models_m (\psi\, \mathcal{S}\, \varphi)$ holds iff for each life cycle in E^l_m containing e there is some event $e' \in Ev^l_m$, with $e' \to^+ e$ such that $M, e', v \models_m \varphi$ holds, and there is a finite chain $\{e_1, \ldots, e_n\} \subseteq Ev_m$ such that $e' \to e_1 \to \ldots \to e_n \to e$ where $M, e_p, v \models_m \psi$ holds for each e_p with $1 \leq p \leq n$,
7. $M, e, v \models_m \Delta\varphi$ holds iff there is an event $e' \in Ev^l_m$ such that $e\ co\ e'$ and $M, e', v \models_m \varphi$ holds,
8. $M, e, v \models_m \varphi \Rightarrow k.\phi$ with $k \neq m$ holds iff $e \in Ev^l_k$ and if $M, e, v \models_m \varphi$ holds then $M, e, v \models_k \phi$ holds,

9. $M, e, v \models_m \varphi \Rightarrow F_k\phi$ with $k \neq m$ holds iff if $M, e, v \models_m \varphi$ holds then there is some $e' \in Ev_k^l$ in each life cycle of E_k such that $e \rightarrow e'$ and $M, e', v \models_k \phi$ holds,
10. $M, e, v \models_m \varphi \Rightarrow P_k\phi$ with $k \neq m$ holds iff if $M, e, v \models_m \varphi$ holds then there is some $e' \in Ev_k^l$ in a life cycle of E_k such that $e' \rightarrow e$ and $M, e', v \models_k \phi$ holds.

Rule 1. defines satisfaction of state formulae for a model M at an event e. A state formula is satisfied by a model M at an event e iff the label of e given by μ^l logically entails σ in the state logic. Due to space limitations we are not referring to satisfaction in P_m^l. Rules 2., 3. and 4. are as usual. Rule 5. gives a weak definition of the until operator. Unlike its usual semantics for sequential life cycles as in [6] we have to consider concurrency within a module life cycle. Figure 5 illustrates the meaning of the operator in a module life cycle. $\varphi\,\mathcal{U}\,\psi$ holds at event e as

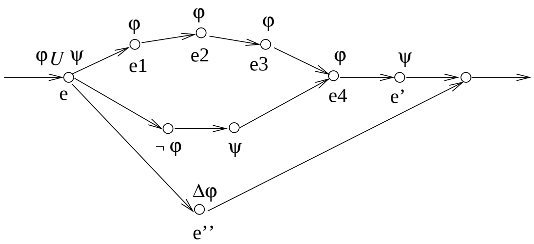

Figure 5: Semantics of the until operator in a module life cycle.

there is an event in the future of e, namely e' where ψ holds, and until then there is a sequence of events where φ holds (e_1, e_2, e_3 and e_4). Other definitions of a stronger until operator can be defined combining the presented weak until with the concurrency operator (cf. below). The rule 6. gives a similar reverse meaning for the since operator $\mathcal{S}$. Rule 7. talks about concurrency. Modules have concurrency and it can be expressed by means of the concurrency operator Δ. In figure 5, $\Delta\varphi$ holds at e'', as there is a concurrent event, for instance e_1, where φ holds. We may additionally define a dual operator for Δ as follows $\nabla\varphi \equiv \neg\Delta\neg\varphi$. The meaning is as follows, $\nabla\varphi$ holds at event e iff for any event e' concurrent to e φ holds. A stronger definition of the until operator may be given by $\varphi\,\mathcal{U}^+\,\psi \equiv (\varphi \wedge \nabla\varphi)\,\mathcal{U}\,\psi$.
Rule 8. defines the meaning of inter-module synchronisation. φ, ϕ are formulae in S_m^l and S_k^l respectively, therefore each of them contains the occurrence of an action (a non asynchronous communication action). These actions synchronise at the shared event e. Rules 9. and 10. reflect inter-module asynchronous communication in both directions. We will explain their semantic with our example.

Example 3.5 To illustrate rule 9., consider event e_3 of the extract of a model given in figure 4. Accordingly to the labelling function given in example 3.3 we know that:

$$M, e_3, v \models_{\texttt{M\_S}} \odot\texttt{Jane.request("ChopinOpus3")} \text{ holds.}$$

Taking the formula φ_3 from example 3.2 and replacing `m` by `"ChopinOpus3"` we may check that it holds at event e_3, i.e.,

$$M, e_3, v \models_{\texttt{M\_S}} \odot\texttt{Jane.request("ChopinOpus3")} \Rightarrow F_{\texttt{B}} \odot \texttt{Bookseller.receive("ChopinOpus3")}$$

In each life cycle of `BOOKSHOP` there is an event corresponding to the receive of Jane's request and such that it is related by causality to e_3 (events g_2 and g_5).
On the other side to illustrate rule 10., we also know that:

$$M, e_{61}, v \models_{\texttt{M\_S}} \odot\texttt{Jane.receive("ChopinOpus3")} \text{ holds.}$$

Again taking formula φ_4 from example 3.2 and replacing `m` by `"ChopinOpus3"` we may check that it holds at event e_{61}, i.e.,

$$M, e_{61}, v \models_{\texttt{M\_S}} \odot\texttt{Jane.receive("ChopinOpus3")} \Rightarrow P_{\texttt{B}} \odot \texttt{Bookseller.deliver("ChopinOpus3")}$$

There is only one life cycle of `BOOKSHOP` containing an event corresponding to the occurrence of the action `deliver("ChopinOpus3")` and such that it is related by causality to e_{61} (event g_4). Moreover, notice that the multiplication of events forced by the construction for concurrent composition guarantees the uniqueness of such a life cycle as in the previous case, and therefore the well-definedness of the rule 10.

Finally, it can be easily seen that for φ_2 from example 3.2 $M, e_{61}, v \models_{\texttt{M\_S}} \varphi_2$ holds. Accordingly to rule 7., there must exist a concurrent event to e_{61} (in this case e_5), where $\odot$`Mary.sing("MadreDeusOMar")` holds. □

4 Discussion

In this paper we presented a modular distributed linear temporal logic and its semantics by means of a small simplified example. The logic is thought for describing dynamic aspects of in-the-large object specification. MDTL is an extension of a distributed temporal logic DTL supporting modularity, intra-module concurrency and inter-module synchronous and asynchronous communication. The asynchronous part of the logic is based on an asynchronous n-agent logic like given in [4]. Since modules are in general not sequential it is necessary to be able to talk about module concurrency. This is achieved by considering a concurrency operator Δ.

Combining both the logic and its underlying model theory allows a whole range of issues like reasoning and verification. Most distributed temporal logics referred in this paper have sound and complete axiomatisations. Therefore, it should be possible to define an axiomatic basis for MDTL which is sound and complete. Nevertheless, reasoning does not require the existence of proof systems and may be done semantically.

Temporal logics developed around the concepts of object-orientation have shown to be very suitable for verification [8, 23]. MDTL also seems to be very promissing for the verification of properties of very large systems due to its modularity and the possibility to move between different module levels. A proof for a statement at a certain level may be done in a top-down manner, i.e., it should be possible to split the proof into smaller ones and carry them out at lower module levels. Then, later on assess if and how the resulting theorems may carry over to higher levels. Since the highest module level corresponds to the system level, global system properties may be expressed. This was a major drawback in DTL where only local object properties were expressable. Comparatively to the *Object Specification Logic* (e.g., [23]), MDTL also allows the uniform treatment of both local and global properties of object systems, and additionally considers modularisation, explicitly expresses concurrency and supports refinement. Moreover, for the presentation of the logic we concentrated on a given module level. No relation of the logics from different levels was explicitly given. Such considerations as well as how to apply the logic for module refinement are subjects for further investigations.

Work reported here is part of an ongoing project on providing a module concept for the specification language TROLL. Comparatively to other existing specification languages, our ideas are very much linked to the concepts available in FOOPS [10], which combines the functional and object-oriented paradigms. However, FOOPS does not support distribution or (a)synchronous communication. The semantics for FOOPS is based on equational logic. A similar approach to FOOPS is also MAUDE [9] MAUDE is based on rewriting logic [18]. Due to the increasing complexity of problem domains, other approaches developed in the last years modularisation concepts for other paradigms, e.g., logic programming. Main ideas in such approaches should be considered, analysed and compared to ours.

It is obviously not enough to provide a language with a well-defined syntax and rigorous semantics for developing large and complex systems. A corresponding CASE-Tool environment has to be provided in order to be able to apply such a language in practice. A WORKBENCH is been developed for the non-modular version of the language TROLL. The architecture and aims of the TROLL-WORKBENCH have been described in [11] among others. The WORKBENCH will contain editors, syntax and semantic checkers, a report generator and an animator. The WORKBENCH should be extended for supporting the modular language. A catalogue for storing reusable modules should also be integrated. Finally, we plan to apply the modular specification language to an industrial case study where we will be able to evaluate the advantages of the approach.

Acknowledgements
I would like to express my gratitude to Hans-Dieter Ehrich for most valuable discussions on object logic. I am specially grateful to Antonio Grau for proof reading and interesting comments. The remarks of the anonymous referees helped to improve the paper.

References

[1] Abadi M. and Lamport L. Conjoining Specifications. *ACM Trans. on Programming Languages and Systems*, 17(3):507–533, May 1995.

[2] Bonner A.J. and Kifer M. Concurrency and Communication in Transaction Logic. In M. Maher, editor, *Proc. 3rd Joint Int. Conf. and Symp. on Logic Programming (JICSLP96), September 2-6 1996, Bonn, Germany*, pages 142–156. MIT Press, 1996.

[3] Ilaria Castellani and Guo-Qiang Zhang. Parallel product of event structures. *Theoretical Computer Science*, 179(1–2):203–215, June 1997.

[4] S. Christensen. A Logical Characterization of Asynchronously Communicating Agents. Technical report, CS Department, Aarhus University, DAIMI PB - 309, 1990.

[5] S. Eckstein. Towards a module concept for object oriented specification languages. In J. Bārzdiņš, editor, *Proc. 3rd Int. Baltic Workshop on Databases and Information Systems, Riga, Latvia, April 15-17,* Volume 2, pages 180–188, 1998.

[6] H.-D. Ehrich, C. Caleiro, A Sernadas, and G. Denker. Logics for Specifying Concurrent Information Systems. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*, pages 167–198. Kluwer Academic Publishers, 1998.

[7] H.-D. Ehrich and A. Sernadas. Local Specification of Distributed Families of Sequential Objects. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Types Specification*, pages 219–235. Springer, Berlin, LNCS 906, 1995.

[8] J. Fiadeiro and T. Maibaum. Temporal Theories as Modularisation Units for Concurrent System Specification. *Formal Aspects of Computing*, 4(3):239–272, 1992.

[9] J. Goguen and J. Meseguer. A Logical Theory of Concurrent Objects and its Realization in the Maude Language. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Direction in Concurrent Object-Oriented Programming*, pages 314–390. The MIT Press, 1993.

[10] J. A. Goguen and A. Socorro. Module Composition and system design for the object paradigm. *Journal of Object-Orientd Programming*, 7(9):47–55, February 1995.

[11] A. Grau, J. Küster Filipe, M. Kowsari, S. Eckstein, R. Pinger, and H.-D. Ehrich. The TROLL Approach to Conceptual Modelling: Syntax, Semantics and Tools. In *Proc. of the 17th Int. Conference on Conceptual Modeling (ER'98), Singapore, November 16-19* , 1998. To appear in LNCS.

[12] P. Hartel, G. Denker, M. Kowsari, M. Krone, and H.-D. Ehrich. Information systems modelling with TROLL formal methods at work. *Information Systems*, 22(2-3):79–99, 1997.

[13] J. Küster Filipe. A Categorical Hiding Mechanism for Concurrent Object Systems. Technical Report 97-06, Technical University Braunschweig, 1997.

[14] J. Küster Filipe. Modelling Parameterisation in Concurrent Object Systems. *Logic Journal of the IGPL*, 5(6):877–879, November 1997. In Conference Report: Workshop on Logic, Language, Information and Computation (WoLLIC)'97, Fortaleza, Brazil, August 20-22.

[15] J. Küster Filipe. Putting Synchronous and Asynchronous Object Modules together: an Event-Based Model for Concurrent Composition. Technical Report 97-05, Technical University Braunschweig, 1997.

[16] J. Küster Filipe. On a Distributed Temporal Logic for Modular Object Systems. Technical Report 98-06, Technical University Braunschweig, 1998.

[17] K. Lodaya, R. Ramanujam, and P.S. Thiagarajan. Temporal Logics for Communicating Sequential Agents. *Int. Journal of Foundations of Computer Science*, 3(2):117–159, 1992.

[18] J. Meseguer. A Logical Theory of Concurrent Objects and its Realization in the Maude Language. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Object-Based Concurrency*. The MIT Press, 1992. Computer System Series.

[19] M. Mukund and P.S. Thiagarajan. An axiomatization of event structures. Technical report, Institute of Mathematical Sciences C.P.T. Campus, Madras 600 113, India, 1989.

[20] M. Nielsen, G. Plotkin, and G. Winskel. Petri Nets, Event Structures and Domains, part 1. *Theoretical Computer Science*, 13:85–108, 1981.

[21] A. Rüping. Modules in object-oriented systems. In R. Ege, M. Sing, and B. Meyer, editors, *TOOLS 1: Technology of Object-Oriented Languages and Systems*. Prentice Hall, 1994.

[22] V. Sassone, M. Nielsen, and G. Winskel. Models for Concurrency: Towards a classification. *Theoretical Computer Science*, 170(1-2):297–348, December 1996.

[23] A. Sernadas, C. Sernadas, and J. Ramos. A temporal logic approach to object certification. *Data & Knowledge Engineering*, 19(3):267–294, June 1996.

[24] C. Sernadas, P. Resende, P. Gouveia, and A. Sernadas. In-the-large object-oriented design of information systems . In F. van Assche, B. Moulin, and C. Rolland, editors, *The Object-oriented Approach in Information Systems* , pages 209—232. North-Holland, 1991.

[25] C. A. Szyperski. Import is Not Inheritance: Why We Need both Modules and Classes. In *Proc. of the European Conference on Object-Oriented Programming* , pages 19–32, 1992.

[26] F.W. Vaandrager. A simple definition for parallel composition of prime event structures. Technical Report CS-R8903, Centre for Mathematics and Computer Science, P.O. Box 4079, 1009 AB Amsterdam, The Netherlands, 1989.