Chapter 2
The Refinement of Multi-Agent Systems

L. Aștefănoaei and F.S. de Boer

Abstract This chapter introduces an encompassing theory of refinement which supports a top-down methodology for designing multi-agent systems. We present a general modelling framework where we identify different abstraction levels of BDI agents. On the one hand, at a higher level of abstraction we introduce the language BUnity as a way to specify “what” an agent can execute. On the other hand, at a more concrete layer we introduce the language BUpL as implementing not only what an agent can do but also “how” the agent executes. At this stage of individual agent design, refinement is understood as trace inclusion. Having the traces of an implementation included in the traces of a given specification means that the implementation is correct with respect to the specification.

We generalise the theory of agent refinement to multi-agent systems in the presence of new coordination mechanisms extended with real time. The generalisation is such that refinement is compositional. This means that refinement at the individual level implies refinement at the multi-agent system level. Compositional property is an important property since it reduces heavily the verification process. Thus having a theory of refinement is a crucial step towards the verification of multi-agent systems’ correctness.
2.1 Introduction

In this chapter we describe a top-down methodology for designing multi-agent systems by refinement. We first focus on the design of individual agents. At this stage, refinement means to reduce the non determinism of high-level agent specification languages. Reducing the non determinism boils down to scheduling policies, i.e., to setting an order (possibly and time) of action executions. The agent specification language we consider is abstract with respect to scheduling policies. It is inspired by UNITY [94], a classical design methodology which emphasises the principles:

- “specify little in early stages of design” and
- “program design at early stages should not be based on considerations of control flow”.

We place ourselves in the framework of BDI models [86]. As already a standard notion, an agent is defined in terms of beliefs, desires, intentions. Beliefs and desires (goals) usually represent the mental state of an agent, and intentions denote the deliberation phase of an agent (often concretising the choice of executing a plan).

We introduce BUnity as an extension of the UNITY language to the BDI paradigm. It is meant to represent an agent in the first stage of design. One only needs to specify initial beliefs and actions (what an agent can do). We make the observation that “specify little” implies non deterministic executions of BUnity agents (actions may be executed in any arbitrary order, for example). On the other hand, BUpL (Belief Update programming Language) enriches BUnity constructions with the notions of plans and repair rules. These are meant to refine the early stage non determinism by specifying how and when actions are executed. In fact, plans implement scheduling policies. We have chosen BUpL as the representation of agents in the last stage of design. Having fixed the levels of abstraction as being BUnity and BUpL, we focus on the correctness of a given BUpL agent with respect to a BUnity specification. By correctness we mean refinement, which is usually understood as trace inclusion. A BUpL agent is correct with respect to a BUnity specification if any possible BUpL behaviour (trace) is also a BUnity one. Since we are interested in applying in practise our methodology, and since verifying trace inclusion is computationally hard, we further focus on simulation as a proof technique for refinement. Additionally, since agents might have infinite behaviours, some of which are unlikely to occur in practice, we provide a declarative approach to modelling fairness and show how simulation works in such a context.

A clear extension of the above framework consists of applying the same methodology to a multi-agent setting. A first step is to lift the notion of abstraction levels from individual agents to multi-agent systems. Considering that the behaviour of the multi-agent system is simply the sum of the behaviours of individual agents is a too unrealistic idea. Instead, we propose action-based coordination mechanisms, to which we refer as choreographies. They represent global synchronisation and ordering conditions restricting the execution of individual agents.
Introducing coordination while respecting the autonomy of the agents is still a challenge in the design of multi-agent systems. However, the advantage of the infrastructures we propose lies in their exogenous feature: the update of the agent’s mental states is separated from the coordination pattern. Nobody changes the agent’s beliefs but itself. Besides that choreographies are oblivious to mental aspects, they control without having to know the internal structure of the agent. For example, whenever a choice between plans needs to be taken, a BUpl agent is free to make its own decision. The degree of freedom can be seen also in the mechanism for handling action failures. The agent chooses one among possibly many available repair rules without being constraint by the choreography. In these regards, the autonomy of agents executed with respect to choreographies is preserved.

Extending the refinement relation from individual agents to multi-agent systems requires solving a new problem since choreographies may introduce deadlocks. It can be that though there is refinement at the individual agent level, adding a choreography deadlocks the concrete multi-agent system but not the abstract one. We take, as example, a choreography which specifies a BUpl agent to execute an action not defined in the agent program itself (but only in the BUunity specification). In this situation, refinement as trace inclusion trivially holds since the set of traces from a deadlocked state is empty. Our methodology in approaching this problem consists of, basically, formalising the following aspects. On the one hand, we define the semantics of multi-agent systems with choreographies as the set of maximal traces, where we make the distinction between a success and a deadlock. These traces consist of the parallel agents’ executions guided by the choreography. We define multi-agent system refinement as maximal trace inclusion. On the other hand, agent refinement becomes ready trace inclusion, where a ready trace records not only the actions being executed, but also those ones which might be executed. We show that multi-agent system refinement is compositional. More precisely, the main result is that agent refinement implies multi-agent system refinement in the presence of any choreography. Furthermore, the refined multi-agent system does not introduce deadlocks with respect to the multi-agent system specification.

The last extension we propose regards time. A more expressive framework can be obtained when action synchronisations depend also on time, not only on the disposal of the agents to perform the actions. Thus, we address the problem of incorporating time into choreographies such that the compositionality result we have remains valid in the timed version. A first step is to extend choreographies by means of timed automata [10] such that they constrain the timings of the actions. Having timed choreographies requires, however, introducing time in agents themselves. Thus, in our case, BUunity and BUpl need to be extended such that they reflect the passing of time. In this respect, we have in mind that basic actions are a common ontology shared by all agents. Since the nature of basic actions does not specify when to be executed, our extension is such that the ontology remains timeless and “when” becomes part of the specific agent applications.

Our contribution consists of introducing a general framework for modelling and not programming agent languages. BUunity and BUpl are simple but expressive
agent languages, inspired by the already standard GOAL [60] and 3APL [222] languages. The operational semantics of the languages makes it easy to prototype them as rewrite systems in Maude [105]. Maude has the advantage that it offers both execution (by rewriting) and verification (by model-checking) of the prototyped systems. We stress the importance of prototyping before implementing complex agent platforms. It is a quick method for proving that the semantics fulfils the initial requirements. We emphasise that all the effort of introducing the formalism of multi-agent system refinement is motivated by the need to perform verification. Multi-agent systems are clearly more complex structures, and their verification tends to become harder. However, in our framework, given the compositionality result, it is enough to verify individual agents in order to conclude properties about the whole multi-agent system.

2.1.1 Related Works

The design methodology we propose integrates in a unifying approach different concepts and results from process theory [196]. Some aspects we deal with have been taken into account in different works, however, from a distinct angle. Considering verification techniques for multi-agent systems, there are already some notable achievements: [73] discusses model-checking AgentSpeak systems, [82] proposes Temporal Trace Language for analysing dynamics between agents, [355] refers to verifying deontic interpreted systems. However, we focus on the compositionality of the refinement relation which reduces the problem of verifying the whole multi-agent system to verifying the agents composing it. Concerning interaction in multi-agent systems, this is usually achieved by means of communication. Communication, in turn, is implemented by message passing, or channel-based mechanisms. This latter can be seen as the basis of implementing coordination artifacts. Such artifacts are usually built in terms of resource access relation in [368], or in terms of action synchronisation [18]. We also mention the language Linda [190] which has not yet been applied in a multi-agent setting but to service oriented services, where the notion of data plays a more important role than synchrony. Control can be also achieved by using social and organisational concepts (e.g., norms, roles, groups, responsibility) and mechanisms (monitoring agents’ actions and sanctioning) [145]. Organisation-based coordination artifacts are recently discussed in [49, 124, 415].

The concepts of choreography and orchestration have already been introduced to web services (to the paradigm Service-oriented Computing), see [25, 311, 317] for different approaches. Though we use the same terminology, our notion of choreography is in essence different since we deliberately ignore communication issues. The choreography model we define is explicit whereas in the other works choreography is implicit in the communication protocol. Thus, we need to take into account deadlock situations that may appear because of “mall-formed” choreographies. Being external, the choreography represents, in fact, contexts while in the other approaches there is a distinction between the modularity and the contextuality of the communi-
cation operator. With respect to timed automata, we mention that its application in a multi-agent system is new. However, timed automata have already been applied to testing real-time systems specifications [214] or to scheduling problems [85].

2.2 From Specification to Implementation Agent Languages

In this section we introduce two agent languages, BUnity and BUpL, each corresponding to different levels of abstractions, with the latter being the more concrete one. We further focus on the correctness of a concrete BUpL agent with respect to a more abstract BUnity agent, where by correctness we understand refinement. In order to automatise such a correctness result we take advantage of simulation as being a sound and complete (under determinacy conditions) proof technique for refinement. Before presenting our methodology, we first recall a few elementary notions from process theory.

2.2.1 Preliminaries

We consider labelled transition systems (LTS) as tuples \((\Sigma, s_0, \text{Act}, \rightarrow)\), where \(\Sigma\) is a finite set of states, \(s_0\) is an initial state, \(\text{Act}\) is a set of actions (labels), and \(\rightarrow\) describes all possible transitions. We denote by \(\tau\) a special action called silent action. \(\text{Act} - \{\tau\}\) is the set of visible actions. We write \(s \xrightarrow{a} s'\) when \((s, a, s') \in \rightarrow\), meaning “\(s\) may become \(s'\) by performing an action labelled \(a\)”. It also means that transition \(a\) is enabled on \(s\). We say that a transition system is deterministic when for any state \(s\), and for any action \(a\), \(s \xrightarrow{a} s'\) and \(s \xrightarrow{a} s''\) implies \(s'' = s'\). We call \(s \xrightarrow{\tau} s'\) an idling transition and we abbreviate it by \(s \xrightarrow{} s'\). The “weak” arrow \(\Rightarrow\) denotes the reflexive and transitive closure of \(\rightarrow\), and \(\xrightarrow{a}\) stands for \(\Rightarrow \xrightarrow{a} \Rightarrow\). A computation in a transition system is defined to be a sequence of the form \(s_0 \xrightarrow{l_0} s_1 \xrightarrow{l_1} s_2 \ldots\), where \(l_i \in \text{Act}, i \in \mathbb{N}\). It can be either finite (when there is no possible transition from the last state), or infinite. For a computation \(\sigma\), the corresponding trace, \(\text{tr}(\sigma)\), is the sequence of visible actions (a word defined on \((\text{Act} - \{\tau\}^\omega)\)). The set of all traces of a system \(S\) (the traces corresponding to all computations starting with the initial state of the system) is denoted by \(\text{Tr}(S)\).

2.2.2 Formalising Mental States and Basic Actions

In the current approach, the underlying logical framework of mental states is a fragment of Herbrand logic. We consider \(F\) and \(\text{Pred}\) infinite sets of function, resp.
predicate symbols, with a typical element \( f \), resp. \( P \). Variables are denoted by the symbol \( x \). Each function symbol \( f \) has associated a non-negative integer \( n \), its arity. Function symbols with 0-arity are also called constants. Terms, usually denoted by the symbol \( t \), are built from function symbols and variables. Formulae, usually denoted by the symbol \( \varphi \), are built from predicates and the usual logical connectors. To sum up, the BNF grammar for terms and formulae is as follows:

\[
\begin{align*}
t & ::= x \mid f(t, \ldots, t) \\
\varphi & ::= P(t, \ldots, t) \mid \neg \varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi
\end{align*}
\]

An atom is any formula \( P(t, \ldots, t) \). A literal is either an atom or the negation of an atom. A term with no variables is called ground. A formula is either ground, or open. In our case, we consider that open formulae have no quantifiers, all variables are understood as being existential. The set of all ground atoms built upon \( F \) and \( Pred \) is a Herbrand model.

Mental states are characterised in terms of beliefs. In the current framework, we consider beliefs as ground atoms, organised in the so-called belief bases (subsets of the Herbrand model), which we denote by \( B \).

Given \( B \) a belief base, the satisfaction relation for ground formulae is defined using structural induction, as usually. For defining the satisfaction relation of open formulae, we consider the usual notion of substitutions as functions that replace variables with terms, denoted by \( [x/t] \ldots [x/t] \). Given a syntactical expression \( e \) and a substitution \( \theta \), we denote the application of \( \theta \) to \( e \) by \( e\theta \), rather than by \( \theta(e) \). The composition \( \theta\theta' \) of two substitutions \( \theta \) and \( \theta' \) is defined as \( \theta\theta'(x) := \theta(\theta'(x)) \), and it is associative. The satisfaction relation for open formulae is as follows:

\[
B \models \varphi \iff \text{exists } \theta \text{ s.t. } B \models \varphi\theta
\]

The substitution \( \theta \) is ground (the substituting terms have no variables). This is because the belief base \( B \) is ground. We make the remark that \( \theta \) is obtained by solving a matching (and not unification) problem. From a complexity point of view, this is important since it is easier to implement a linear algorithm for matching than for unification. We say a term \( s \) matches a ground term \( t \) if there exists a substitution \( \theta \) such that \( s\theta \) is syntactically equal to \( t \). The matching problem extends easily to formulae and belief bases. We consider \( Sols(B \models \psi) \) as the set of all matchers. We say the substitution is the identity function when \( \psi \) is a ground formula satisfied in \( B \).

Basic actions are functions defined as pairs \((\psi, \xi)\), where \( \psi \) are formulae which we call preconditions, and \( \xi \) are sets of literals which we call effects. The following inclusions are required:

\[
\text{Var}(\xi) \subseteq \text{Var}(\psi) = \{x_1, \ldots, x_n\},
\]

where \( \text{Var}(e) \) denotes the set of variables in a syntactic expression \( e \). We use the symbol \( A \) for the set of basic actions’ definitions. We refer to \( Act \) as the set of basic action names with typical element \( a \). We further use the notation \( a\theta \) to represent
action terms which result from the application of substitutions. By abuse of notation we sometimes refer to $a(\bar{x})$ (resp. $\psi(\bar{x})$, $\xi(\bar{x})$) as $a$ (resp. $\psi$, $\xi$) when $\bar{x}$, the set of variables, is not relevant.

Given a basic action definition $a = (\psi, \xi)$, if matching $\psi$ to $B$ has a solution $\theta$, then the effect of $a\theta$ is to update the belief base by adding or removing ground atoms from the set $\xi\theta$:

$$\begin{align*}
B \cup l\theta &= B \cup l\theta, \quad l \in \xi \\
B \cup \neg l\theta &= B \setminus l\theta, \quad \neg l \in \xi
\end{align*}$$

We write $B \cup \xi\theta$ to represent the result of an update operation, which is automatically guaranteed to be consistent since we add only positive literals.

### 2.2.3 BUnity Agents

BUnity language represents abstract agent specifications. Its purpose is to model agents at a coarse level, using a minimal set of constructions. A BUnity agent abstracts from specific orderings (for example, action planning). Hence her executions are highly non deterministic.

The mental state of a BUnity agent is simply a belief base. On top of basic actions, Bunity language allows a finer type of construction, conditional actions, which are organised in a set denoted by $C$. A conditional action is built upon a basic action. It is syntactically defined by $\phi \triangleright do(a)$, where $\phi$ is a query on the belief base, and $a$ is the name of an action. Intuitively, conditional actions are like `await` statements in imperative languages: `await $\phi$ do $a$`, action $a$ can be executed only when $\phi$ matches the current belief base.

Both basic and conditional actions have enabling conditions, and this might raise confusion when distinguishing them. The intuition lying behind the need to consider them both is that they demand information at different levels. The precondition of a basic action should be understood a built-in restriction which internally enables belief updates. It is independent of the particular choice of application. A conditional action is executed in function of certain external requirements (reflected in the mental state). Thus it is application dependent. The external requirements are meant to be independent of the built-in enabling mechanism. Whether is agent “$A$” or agent “$B$” executing an action $a$, the built-in condition should be the same for both of them. Nevertheless, each agent may have its own external trigger for action $a$.

A BUnity agent is defined as a tuple, $(B_0, A, C)$, where $B_0$ is a set of initial beliefs. For such a configuration, we define an operational semantics in terms of labelled transition systems.

**Definition 2.1 (BUnity Semantics).** Let $(B_0, A, C)$ be a BUnity configuration. The associated LTS is $(\Sigma, B_0, L, \rightarrow)$, where:
\* \( \Sigma \) is a set of states (belief bases)
\* \( \mathcal{B}_0 \) is the initial state (the initial belief base)
\* \( L \) is a set of ground action terms
\* \( \rightarrow \) is the transition relation, given by the rule:

\[
\phi \Rightarrow \text{do}(a) \in C \quad a = (\psi, \xi) \in \mathcal{A} \quad \theta \in \text{Sols}(\mathcal{B} \models (\phi \land \psi)) \quad (\text{act})
\]

\[
\mathcal{B} \xrightarrow{a_\theta} \mathcal{B} \cup \xi\theta
\]

We consider the meaning of a BUnity agent defined in terms of possible sequences of mental states, and its externally observable behaviour as sequences of executed actions. Equally, the meaning of a BUnity agent is the set of all possible computations of its associated LTS, and the behaviour, the trace set (as words on action terms). Our focus on visible actions (and not states) is motivated by the fact that, in studying simulation, we are interested in what we see and not how the agent thinks. We take the case of a robot: one simulates his physical actions, lifting or dropping a block, for example, and not the mental states of the robot.

The transition rule \((\text{act})\) captures the effects of performing the action \(a\). It basically says that if there is a conditional action \(\phi \Rightarrow \text{do}(a)\) and the query \(\phi\) has a solution in the current mental state then if the precondition of \(a\) matches the current belief base new beliefs are added/deleted with respect to the effects of \(a\).

We take, as an illustration, a known problem, which we first found in [222], of an agent building a tower of blocks. An initial arrangement of three blocks \(A, B, C\) is given there: \(A\) and \(B\) are on the floor, and \(C\) is on top of \(A\). The goal of the agent is to rearrange them such that \(A\) is on the floor, \(B\) on top of \(A\) and \(C\) on top of \(B\). The only action an agent can execute is to move one block on the floor, or on top of another block, if the latter is free.

\[
\begin{align*}
\mathcal{B}_0 &= \{ \text{on}(C,A), \text{on}(A,\text{floor}), \text{on}(B,\text{floor}), \\
& \quad \text{free}(B), \text{free}(C), \text{free}(\text{floor}) \} \\
\mathcal{A} &= \{ \text{move}(x,y,z) = (\text{on}(x,y) \land \text{free}(x) \land \text{free}(z), \\
& \quad \{ \text{on}(x,z), \neg \text{on}(x,y), \neg \text{free}(z) \} \} \}
\end{align*}
\]

\[
C = \{ \neg (\text{on}(B,A) \land \text{on}(C,B)) \Rightarrow \text{do}(\text{move}(x,y,z)) \}
\]

Fig. 2.1 A BUnity Toy Agent

The example from Figure 2.1 is taken in order to underline the difference between enabling conditions (for basic actions) and triggers (for conditional actions): on the one hand, it is possible to move a block \(x\) on top of another block \(z\), if \(x\) and \(z\) are free; on the other hand, given the goal of the agent, moves are allowed only when the configuration is different than the final one.

\footnote{We do not explicitly model goals. Please check Section 2.2.7 for a discussion motivating our choice.}
2.2.4 Why BUnity Agents Need Justice

In non deterministic systems that abstract from scheduling policies, some traces are improbable to occur in real computations. In this sense, the operational semantics (given by transition rules) is too general in practise: if the actions an agent can execute are always enabled, it should not be the case that the agent always chooses the same action. Such executions are usually referred to as being unfair.

For example, we imagine a scenario illustrative for cases where modelling fairness constraints is a “must”. For this, we slightly complicate the “tower” problem from the previous section, by giving the agent described in Figure 2.1 an extra assignment to clean the floor, if it is dirty. Thus the agent have two alternatives: either to clean or to build. We add a basic action, $\text{clean} = (\neg \text{cleaned}, \{\text{cleaned}\})$. We enable the agent to execute this action at any time, by setting $\top$ as the query of the conditional action calling $\text{clean}$, i.e., $\top \downarrow \text{do}(\text{clean})$.

We note that it is possible that the agent always prefers cleaning the floor instead of rearranging blocks, in the case that the floor is constantly getting dirty. We want to cast aside such traces and moreover, we want a declarative, and not imperative solution. Our option is to follow the approach from [296]: we constrain the traces by adding fairness conditions, modelled as linear temporal logic (LTL) properties. Fairness is there expressed either as a weak, or as a strong constraint. They both express that actions which are “many times” enabled on infinite execution paths should be infinitely often taken. The difference between them is in the definition of “many times” which is continuously (resp. infinitely often). Due to the semantics of conditional actions, it follows that the choice of executing one action cannot disable the ones not chosen and thus BUnity agents only need weak fairness.

Definition 2.2 (Justice [296]). A trace is just (weakly fair) with respect to a transition $a$ if it is not the case that $a$ is continually enabled beyond some position, but taken only a finite number of times.

To model such a definition as LTL formulae we need only two future operators, $\diamond$ (eventually) and $\Box$ (always). Their satisfaction relation is defined as follows:

$\sigma \models \diamond \phi$ iff $(\exists i > k)(s_i \models \phi)$

$\sigma \models \Box \phi$ iff $(\forall i > k)(s_i \models \phi)$,

where $s_0, \ldots, s_k, \ldots$ are the states of a computation $\sigma$. By means of these operators we define weak fairness for BUnity as:

$$\text{just}_1 = \bigwedge_{a \in \text{Act}} (\diamond \Box \text{enabled}(\phi \uparrow \text{do}(a)) \rightarrow \Box \diamond \text{taken}(a)) .$$

where enabled and taken, predicates on the states of BUnity agents, are defined as:

$\mathcal{B} \models \text{enabled}(\phi \uparrow \text{do}(a))$ iff $a = (\psi, \xi) \land \mathcal{B} \models \phi \land \psi$

$\mathcal{B} \models \text{taken}(a)$ iff $\mathcal{B} \overset{a\theta}{\rightarrow} \mathcal{B}'$
Such a fairness condition ensures that all fair BUnity traces are of the form \((clean^n (move\theta)^m)^\omega\), or equally \([\forall n,m \in \mathbb{N}, k \in \mathbb{N} \cup \{\infty\}]\). We note that the advantage of a declarative approach to modelling fairness is the fact that we do not need to commit to a specific scheduling policy as it is the case when implementing fairness by means of a scheduling algorithm, e.g., Round-Robin. A scheduling policy would basically correspond to fixing the exponents \(n\) and \(m\).

### 2.2.5 BUpL Agents

The BUnity agent described in Figure 2.1 is highly non deterministic. It is possible that the agent moves \(C\) on the floor, \(B\) on \(A\), and \(C\) on \(B\). This sequence represents, in fact, the shortest one to achieving the goal. However, it is also possible that the agent pointlessly move \(C\) from \(A\) to \(B\) and then back from \(B\) to \(A\).

BUpL language allows the construction of plans as a way to order actions. We refer to \(\mathcal{P}\) as a set of plans, with a typical element \(p\), and to \(\Pi\) as a set of plan names, with a typical element \(\pi\). Syntactically, a plan is defined by the following BNF grammar:

\[
p ::= a(t_1, \ldots, t) | \pi(t_1, \ldots, t) | a(t_1, \ldots, t); p | p + p
\]

with ‘;’ being the sequential composition operator and ‘+’ the choice operator, with a lower priority than ‘;’.

The construction \(\pi(x_1, \ldots, x_n)\) is called abstract plan. It is a function of arity \(n\), defined as \(\pi(x_1, \ldots, x_n) = p\). Abstract plans should be understood as procedures in imperative languages: an abstract plan calls another abstract plan, as a procedure calls another procedure inside its body.

BUpL language provides a mechanism for handling the failures of actions in plans through constructions called repair rules. A plan fails when the current action cannot be executed. Repair rules replace such a plan with another. Syntactically, they have the form \(\phi \leftarrow p\), and it means: if \(\phi\) matches \(\mathcal{B}\), then substitute the plan that failed for \(p\).

A BUpL agent is a tuple \((\mathcal{B}_0, \mathcal{A}, \mathcal{P}, \mathcal{R}, p_0)\), where \(\mathcal{B}_0, \mathcal{A}\) are the same as for a BUnity agent, \(p_0\) is the initial plan, \(\mathcal{P}\) is a set of plans and \(\mathcal{R}\) is a set of repair rules.

Plans, like belief bases, have a dynamic structure, and this is why the mental state of a BUpL agent incorporates both the current belief base and the plan in execution. The operational semantics for a BUpL agent is as follows:

**Definition 2.3 (BUpL Semantics).** Let \((\mathcal{B}_0, \mathcal{A}, \mathcal{P}, \mathcal{R}, p_0)\) be a BUpL configuration. Then the associated LTS is \((\Sigma, (\mathcal{B}_0, p_0), L, \rightarrow)\), where:

- \(\Sigma\) is a set of states, tuples \((\mathcal{B}, p)\)
- \((\mathcal{B}_0, p_0)\) is the initial state
$p = (a;p') \quad a = \psi, \xi \in \mathcal{A} \quad \theta \in \text{Sols}(\mathcal{B} \models \psi)$ (act)

$\mathcal{B}, p \xrightarrow{\mu} \mathcal{B}', p'$ (sum_i)

$\mathcal{B}, a; p \xrightarrow{\mu} \phi \leftarrow p' \in \mathcal{R}$ (fail)

Fig. 2.2 BUpL Rules

- $L$ is a set of labels, either ground action terms or $\tau$
- $\rightarrow$ represents the transition rules given in Figure 2.2.

As it was the case for BUnity agents, we consider the meaning of a BUpL agent defined in terms of possible sequences of mental states, and its externally observable behaviour as sequences of executed actions.

The transition rule (act) captures the effects of performing the action $a$ which is the head of the current plan. It basically says that if $\theta$ is a solution to the matching problem $\mathcal{B} \models \psi$ where $\psi$ is the precondition of action $a$ then the current mental state changes to a new one, where the current belief base is updated with the effects of $a$ and the current plan becomes the “tail” of the previous one. The transition rule (fail) handles exceptions. If the head of the current plan is an action that cannot be executed (the set of solutions for the matching problem is empty) and if there is a repair rule $\phi \leftarrow p'$ such that the new matching problem $\mathcal{B} \models \phi$ has a solution $\theta$ then the plan is replaced by $p'$. The transition rule ($\pi$) implements “plan calls”. If the abstract plan $\pi(x_1, ..., x_n)$ defined as $p(x_1, ..., x_n)$ is instantiated with the terms $t_1, ..., t_n$ then the current plan becomes $p(t_1, ..., t_n)$ which stands for $p[x_1/\tau_1] ... [x_n/\tau_n]$. The transition rule (sum_i) replaces a choice between two plans by either one of them. The label $\mu$ can be either a ground action name or a $\tau$ step, in which case $\mathcal{B}' = \mathcal{B}$, and $p'$ is a valid repair plan (if any).

We take as an example a BUpL agent that solves the tower of blocks problem. It has the same initial belief base and the same basic action as the BUnity agent.

The BUpL agent from Figure 2.3 is modelled such that it illustrates the use of repair rules: we explicitly mimic a failure by intentionally telling the agent to move $B$ on $A$. Similar scenarios can easily arise in multi-agent systems: imagine that initially $C$ is on the floor, and the agent decides to move $B$ on $A$; imagine also that another
\( \mathcal{B}_0 = \{\ on(C,A),\ on(A,floor),\ on(B,floor),\ free(B),\ free(C),\ free(floor) \} \)

\( \mathcal{A} = \{\ move(x,y,z) = (on(x,y) \land \ free(x) \land \ free(z),\ on(x,z),\ \lnot on(x,y),\ \lnot free(z) \} \} \)

\( \mathcal{P} = \{\ p_0 = move(B,floor,A); move(C,floor,B) \} \)

\( \mathcal{R} = \{\ on(x,y) \leftarrow move(x,y,floor); p_0 \} \)

Fig. 2.3 A BUpL Toy Agent

agent comes and moves \( C \) on top of \( A \), thus moving \( B \) on \( A \) will fail. The failure is handled by \( on(x,y) \leftarrow move(x,y, floor); p_0 \). Choosing \( [x/A][y/C] \) as a matcher, enables the agent to move \( C \) on the floor and after the initial plan can be restarted.

2.2.6 Why BUpL Agents Need Compassion

Though BUpL agents are meant to reduce the non determinism from BUnity agent specifications, unfair executions are not ruled out because of the non determinism in the choices between plans and/or repair rules. To illustrate this, we assign a mission plan to the BUpL agent described in Figure 2.3, \( mission = cleanR + rearrange(B,A,C) \), where \( cleanR \) is a tail-recursive plan, \( cleanR = clean; cleanR \), with \( clean \) being the action defined in Section 2.2.4. The plan \( rearrange \) generalises the previously defined \( p_0 \): \( rearrange(x,y,z) = move(x,floor,y); move(z,floor,x) \). It consists of reorganising free blocks placed on the floor, such that they form a tower. This plan fails if not all the blocks are on the floor, and the failure is handled by the already defined repair rule, which we call \( r_1 \). We add a repair rule, \( r_2, T \leftarrow mission \), which simply makes the agent restart the execution of the plan \( mission \).

As it was the case with the BUnity agent from the Section 2.2.4, it is possible, in the above scenario, that the BUpL agent always prefers cleaning the floor instead of rearranging blocks, though this is useless when the floor has already been cleaned. Nevertheless, such cases are disregarded if one requires that executions are fair. The only difference from the fairness condition imposed on the executions of BUnity agents is that plans need not be continuously but infinitely often enabled.

We consider two scenarios for defining fairness with respect to choices in repair rules and plans. The execution of \( rearrange \) has failed. Both repair rules \( r_1 \) and \( r_2 \) are enabled, and always choosing \( r_2 \) makes it impossible to make the rearrangement. This would not be the case if \( r_1 \) were triggered. It follows that the choice of repair rules should be weakly fair:

\[
just_2 = \bigwedge_{p \in \mathcal{P}} (\Diamond \Box enabled(\phi \leftarrow p) \rightarrow \Box \Diamond taken(p)) .
\]
The repair rule $r_1$ has been applied, and all three blocks are on the floor. Returning to the initial mission and being in favour of cleaning leads again to a failure (the floor is already clean). The only applicable repair rule is $r_2$ which simply tells the agent to return to the mission. Thus, it can be the case that, though rearranging the blocks is enabled, it will never happen, since the choice goes for the plan clean (which always fails). Therefore, because plans are not continuously enabled, their choice has to be strongly fair:

**Definition 2.4 (Compassion [296]).** A trace is compassionate (strongly fair) with respect to a transition $a$ if it is not the case that $a$ is infinitely often enabled beyond some position, but taken only a finite number of times.

As it was the case with justice, modelling the above definition as a linear temporal logic formula is straightforward, however we refer to plans instead of actions:

$$compassionate = \bigwedge_{p \in P} (\Box \Diamond enabled(p) \rightarrow \Box \Diamond taken(p))$$

In the above scenarios enabled and taken are defined similarly as in the case of actions for the language BUnity: (1) a repair rule is enabled when its precondition is satisfied in the belief base; (2) a plan is enabled when the precondition of its first action is satisfied; (3) a plan is taken when its first action is taken.

$$(\mathcal{B},a;p) \models enabled(a;p) \text{ iff } a = (\psi,\xi) \land \mathcal{B} \models \psi$$

$$(\mathcal{B},a;p) \models enabled(\phi \leftarrow p') \text{ iff } \mathcal{B} \models \phi$$

$$(\mathcal{B},a;p) \models taken(a;p) \text{ iff } (\mathcal{B},a;p) \xrightarrow{a} (\mathcal{B}',p)$$

The fairness conditions ensure that all fair BUpL traces are of the form $(clean^* (move\theta)^*)^\omega$ which is exactly the same as in the case of the BUnity agent. This is a positive result, since we are interested in the fair refinement between the BUpL and the BUnity agent.

### 2.2.7 Appraising Goals

We have deliberately cast aside goals in BUnity and BUpL. This is mainly for simplicity reasons. The usual way ([60, 222]) to explicitly incorporate goals is to fix a particular representation, for example, as a conjunction of ground atoms (which we might understand as a special case of a belief base). The corresponding change in the semantics is to extend the queries of BUnity conditional actions and of BUpL repair rules such that they do not interrogate only belief bases but also goals. Additionally, plan calls should be extended such that goals can trigger plan executions.

Given that our focus is on verification, being able to represent goals implicitly is acceptable enough in our framework. Furthermore, the expressive power of the
languages is not necessarily decreased. We can, without changing the syntax and the semantics of the languages, have a declarative modelling of goals as LTL formulae. In such a situation, we would be interested in any agent execution which satisfies a given goal. This problem can be equally stated as a reachability problem and the answer can be provided by verification. More precisely, model-checking the negation of the goal returns, in fact, a counter-example denoting a successful trace (leading to the achievement of the goal) in the case that there exists one.

For example, we can define, with respect to the scenario introduced in the previous sections, the LTL predicates

\[ \text{goal}_1 = \Diamond \text{fact}(\text{cleaned}) \]
\[ \text{goal}_2 = \Diamond (\text{fact}(\text{on}(\text{A}, \text{floor})) \land \text{fact}(\text{on}(\text{B}, \text{A})) \land \text{fact}(\text{on}(\text{C}, \text{B}))) \]

where \text{fact} is a predicate defined on the mental states of either BUnity or BUpL agents in the following way:

\[ (\mathcal{B}, p) \models \text{fact}(P) \text{ iff } \mathcal{B} \models P. \]

Model-checking that the property \( \neg (\text{goal}_1 \land \text{goal}_2) \) holds in a state reachable from the initial one returns a counterexample representing the minimal trace \text{clean} \text{move}(\text{C}, \text{floor}) \text{move}(\text{B}, \text{A}) \text{move}(\text{C}, \text{B}). This execution leads to a state where both \text{goal}_1 and \text{goal}_2 are satisfied.

### 2.3 The Refinement of Individual Agents

We have already mentioned in the introduction that we understand BUnity as a typical abstract specification language, and BUpL as an implementation language. Since control aspects are ignored in BUnity models, BUnity is a “highly” non deterministic language. Such non determinism is reduced in BUpL agents. We are interested in the correctness of a BUpL agent with respect to a BUnity agent, or equally stated, in the refinement between a BUpL and a BUnity agent. Refinement is usually defined as trace inclusion, all the traces of the implementation are contained among the traces of the specification.

**Definition 2.5 (Refinement).** Let \((\mathcal{B}_0, \mathcal{A}, C)\) be a BUnity agent with its initial mental state \(\mathcal{B}_0\) and let \((\mathcal{B}_0, \mathcal{A}, P, R, p_0)\) be a BUpL agent with its initial mental state \((\mathcal{B}_0, p_0)\). We say that the fair executions of the BUpL agent refine the fair executions of the BUnity agent \((\mathcal{B}_0, p_0) \subseteq \mathcal{B}_0\) if and only if every trace of the BUpL agent is also a trace of the BUnity agent, that is \(Tr((\mathcal{B}_0, p_0)) \subseteq Tr(\mathcal{B}_0)\).

Being that we are interested only in fair agent executions, we consider also refinement in terms of fair trace inclusion where we take into account the definitions of \text{just}_1, \text{just}_2 and \text{compassionate} as introduced in the previous sections.
Definition 2.6 (Fair Refinement). Let \((B_0, \mathcal{A}, C)\) be a BUnity agent with its initial mental state \(B_0\) and let \((B_0, \mathcal{A}, \mathcal{P}, \mathcal{R}, p_0)\) be a BUpL agent with its initial mental state \((B_0, p_0)\). We say that the fair executions of the BUpL agent refine the fair executions of the BUnity agent \((B_0, p_0) \subseteq f B_0\) i f every fair trace of the BUpL agent is also a fair trace of the BUnity agent, that is \((\forall tr \in Tr((B_0, p_0))) (\sigma_{tr} \models \text{compassionate} \land \text{just}2) \Rightarrow (tr \in Tr(B_0) \land \sigma'_{tr} \models \text{just}1)\), where \(\sigma_{tr}\) (resp. \(\sigma'_{tr}\)) is any corresponding computation path in the transition system associated to the BUpL (resp. BUnity) agent.

We note that in the above definitions we have used the same symbols for both initial belief bases \((B_0)\) and sets of action definitions \((\mathcal{A})\). This is not a restriction, it only simplifies the notation.

Proving refinement by definition is not practically feasible because the set of traces may be considerably large. We would need to check that for any solution to matching problems the corresponding trace belongs to both implementation and specification. Instead, refinement is usually proved by means of simulation, which has the advantage of locality of search: one looks for checks at the immediate (successor) transitions that can take place. We recall that the possible transitions for BUpL and BUnity agents are either \(\tau\) steps (corresponding to choices between plans and repair rules) or steps labelled with action terms. Since we are interested in simulating only visible actions, we refer to weak simulation, which is oblivious with respect to \(\tau\) steps.

Definition 2.7 (Weak Simulation). Let \((B_0, \mathcal{A}, C)\) be a BUnity agent with its initial mental state \(B_0\) and let \((B_0, \mathcal{A}, \mathcal{P}, \mathcal{R}, p_0)\) be a BUpL agent with its initial mental state \((B_0, p_0)\). Let \(\Sigma, \Sigma'\) be the sets of mental states for each agent \((B_0 \in \Sigma, (B_0, p_0) \in \Sigma')\) and let \(R\) be a relation, \(R \subseteq \Sigma \times \Sigma'\). \(R\) is called a weak simulation if, whenever \(B_0 R (B_0, p_0)\), if \((B_0, p_0) \xrightarrow{a} (B, p)\), then it is also the case that \(B_0 \xrightarrow{a} B\) and \(B R (B, p)\).

Definition 2.8. Let \((B_0, \mathcal{A}, C)\) be a BUnity agent with its initial mental state \(B_0\) and let \((B_0, \mathcal{A}, \mathcal{P}, \mathcal{R}, p_0)\) be a BUpL agent with its initial mental state \((B_0, p_0)\). We say that the BUnity agent weakly simulates the BUpL agent \((B_0, p_0) \preceq B_0\) if there exists a weak simulation \(R\) such that \(B_0 R (B_0, p_0)\).

We recall that in general simulation is a sound but not necessarily complete proof technique for refinement. We take the classical situation from Figure 2.4 as a counter-example. However, simulation is complete when the simulating system is deterministic (see, for example, [196]). We make the remark that in the case of finite transition systems it is always possible to determinise a non deterministic system by means of a power set construction ( [9] for the case of finite traces, and [319, 382] for the case of infinite traces). However, “determinization” is computationally hard \((2^{O(n\log n)})\) in the number of states [382]) and thus usually unfeasible when the focus is on verification.

In our case, the simulating agent is a BUnity one. BUnity agents, though highly non deterministic with respect to control issues, have the property that they are modelled by deterministic (see the definition from Section 2.2.1) transition systems. This
is because though a BUnity agent makes arbitrary decisions regarding which action to execute, the mental state reflecting the effect of the chosen action is uniquely determined, thus actions themselves are deterministic. It follows that, in our framework, simulation is not only a sound but also a complete proof technique for refinement.

We want to reduce the problem of deciding simulation between a BUUpL and a BUnity agent to a verification problem. For this, we give a modal characterisation to simulation by constructing the synchronised product of a BUUpL and BUnity agent and by defining an LTL property on the states of the product. The property is basically satisfied when the product reaches a deadlock state. We show that it is sufficient and necessary to detect the existence of a deadlocked state in order to prove simulation, and thus refinement.

**Definition 2.9 (BUUpL-BUnity Synchronised Product).** Let $(B_0, A, C)$ be a BUnity agent with its initial mental state $B_0$ and let $(B_0, A, P, R, p_0)$ be a BUUpL agent with its initial mental state $(B_0, p_0)$. If $I = (\Sigma, (B_0, p_0), Act \cup \{\tau\}, \rightarrow_1)$ and $S = (\Sigma', B_0, Act, \rightarrow_2)$ are the corresponding transition systems to the BUUpL, resp. BUnity agent, then their left synchronised product is $I \otimes S = (\Sigma \times \Sigma', (B_0, p_0), B, \rightarrow)$. The semantics is given by the following transition rule:

$$
\frac{(B, p) \xrightarrow{a} (B', p')} \quad B \rightarrow_2 B'}
\langle\langle (B, p), B \rangle \xrightarrow{a} \langle\langle (B', p'), B' \rangle
$$

Mathematically, the choice between either first the BUUpL agent performs the step and then the BUnity performs the same step or the other way around is insignificant. However, from an implementation point of view, it is better to make the transition rule conditional. Only if the BUUpL agent can fire an action the product changes state depending on whether the BUnity agent can mimic the BUUpL agent. We say that the BUUpL agent drives the simulation. We further say that if the BUnity agent can execute the same action, the product reaches a “good” state. Otherwise, the product is in a deadlocked state.

**Definition 2.10 (Deadlock).** Let $\perp$ be the property $(\langle (B, p), B \rangle \xrightarrow{a} (B', p') \land B'' \xrightarrow{\mu} 2)$. The state $\langle (B, p), B \rangle$ has a deadlock when $\perp$ holds. That is:

$$
\langle (B, p), B \rangle \models \perp \text{ iff } (B, p) \xrightarrow{a} (B', p') \land B'' \xrightarrow{\mu} 2.
$$
We say that the product is *deadlock-free* if it has no deadlocks.

We note that we make the difference between a deadlocked and a terminal product state, where the only possible transition for the BUpL agent is the idling transition. We further make the remark that, since it basically depends on the BUnity agent being able to perform a certain action, the definition of deadlock introduces *asymmetry* in the executions of the BUpL and BUnity product.

**Proposition 2.1.** Let $\langle B_0, A, C \rangle$ be a BUnity agent with its initial mental state $B_0$ and let $\langle B_0, A, P, R, p_0 \rangle$ be a BUpL agent with its initial mental state $(B_0, p_0)$. We then have that the BUpL agent refines the BUnity agent ($B_0, p_0 \subseteq B_0$) if and only if $\langle (B_0, p_0), B_0 \rangle \models \Box \neg \bot$, where $\langle (B_0, p_0), B_0 \rangle$ is the initial state of the BUpL-BUnity left synchronised product.

**Proof.** Since the proof is basically a simplification of the one we present for Theorem 2.1, we leave it to the reader.

In what follows we focus on the “fair” version of Proposition 2.1.

**Theorem 2.1.** Let $\langle B_0, A, C \rangle$ be a BUnity agent with its initial mental state $B_0$ and let $\langle B_0, A, P, R, p_0 \rangle$ be a BUpL agent with its initial mental state $(B_0, p_0)$. We then have that the BUpL agent fairly refines the BUnity agent ($B_0, p_0 \subseteq f B_0$) if and only if $\langle (B_0, p_0), B_0 \rangle \models \text{compasionate} \land \text{just}_2 \rightarrow \text{just}_1 \land \Box \neg \bot$, where $\langle (B_0, p_0), B_0 \rangle$ is the initial state of the BUpL-BUnity left synchronised product.

**Proof.** We recall that an LTL property holds in a state $s$ if and only if it holds for any computation path $\sigma$ beginning with $s$.

"$\Rightarrow$": Assume that $\langle (B_0, p_0), B_0 \rangle \not\models \text{compasionate} \land \text{just}_2 \rightarrow \text{just}_1 \land \Box \neg \bot$. This means that there exists a computation path $(\sigma, \sigma')$ in the BUpL-BUnity product such that $\sigma \models \text{compasionate} \land \text{just}_2$ (*) and either (1) $\Diamond \bot$ or (2) $\neg \text{just}_1$ holds. From (*) we have that $tr(\sigma)$ is a fair BUpL trace. From the hypothesis $(B_0, p_0) \subseteq f B_0$ we have that there exists a BUnity computation path (which must be $\sigma'$ since BUnity is deterministic) such that $tr(\sigma) = tr(\sigma')$ and $\sigma' \models \text{just}_1$ thus (2) cannot be true. Let us now consider (1). In order to have that $\Diamond \bot$ holds for $(\sigma, \sigma')$ there must be a deadlocked state $\langle (B, p), B \rangle$ on this path. But this implies that there is a fair BUpL trace $tr(\sigma)a$ which does not belong to the set of fair BUnity traces, thus contradicting the hypothesis.

"$\Leftarrow$": Assume that $\langle (B_0, p_0), B_0 \rangle \models \text{compasionate} \land \text{just}_2$, meaning that any product path is fair with respect to the BUpL path. We make the remark that we do not need to worry about the “vacuity” problem (compasionate $\land$ just being always false) since there always exists a fair computation path (any scheduling algorithm will provide one). We now need to prove that $\langle (B_0, p_0), B_0 \rangle \models \text{just}_1$ (*) and $\langle (B_0, p_0), B_0 \rangle \models \Box \neg \bot$ (**) implies fair refinement. We do this by proving that (**) implies simulation (thus
also refinement). Since we have (*), the product paths are also fair with respect to BUnity.

We now construct a relation $R$ containing all pairs of BU_{UpL} and BU_{Unity} reachable states and we prove $R$ is a simulation relation. Let $R = \{(\mathcal{B}, p) \mid \langle (\mathcal{B}_0, p_0), \mathcal{B}_0 \rangle \rightarrow^* \langle (\mathcal{B}, p), \mathcal{B} \rangle \}$. Let $((\mathcal{B}, p), \mathcal{B}) \in R$ s.t. $(\mathcal{B}, p) \mathcal{A} \rightarrow (\mathcal{B}', p')$. It is then the case that also $\mathcal{B} \rightarrow \mathcal{B}'$ otherwise $\langle (\mathcal{B}, p), \mathcal{B} \rangle \models \perp$. We further need to prove that $\langle (\mathcal{B}', p), \mathcal{B}' \rangle$ is in $R$. This is, indeed, true since $\langle (\mathcal{B}_0, p_0), \mathcal{B}_0 \rangle \rightarrow^* \langle (\mathcal{B}, p), \mathcal{B} \rangle \mathcal{A} \rightarrow \langle (\mathcal{B}', p), \mathcal{B}' \rangle$ thus $\langle (\mathcal{B}', p), \mathcal{B}' \rangle$ is a reachable state of the product. \[\Box\]

**Remark 2.1.** Refinement does not necessarily imply fair refinement. We consider a BU_{UpL} agent which can continuously perform only action $a$ while the BU_{Unity} specification can additionally perform $b$. Refinement trivially holds ($\{a^\omega\} \subset \{(a^*b^*)^\omega\}$) however $a^\omega$ is unfair with respect to BU_{Unity}.

We take, for example, the BU_{UpL} and BU_{Unity} agents building the ABC tower. Any visible action that BU_{UpL} executes can be mimicked by the BU_{Unity} agent, thus in this case BU_{Unity} simulates BU_{UpL} and refinement is guaranteed. If we now pose the question whether fair executions of the BU_{UpL} agent refine fair executions of the BU_{Unity} agent, we have that, conforming to Theorem 2.1, the answer is positive if the formula $(\text{compassionate} \land \text{just}_2) \rightarrow (\text{just}_1 \land \square \neg \perp)$ is satisfied in the left product. This is because the traces of the product are of the form $(\text{clean}^* (\text{move}^\theta)^*)^\omega$ and thus satisfying the fairness constraints for both BU_{UpL} and BU_{Unity}.

### 2.4 Towards Multi-Agent Systems

If previously it was enough to refer to an agent by its current mental state, this is no longer the case when considering multi-agent systems. This is why we associate with each agent an identifier and we consider a multi-agent system as a finite set of such identifiers. We further denote a state of a multi-agent system by $M = \{(i, ms_i) \mid i \in I\}$, where $I$ is the set of agent identifiers and $ms_i$ is a mental state for the agent $i$. For the moment, we abstract from what is the mental state of an agent. The choice of representation is not relevant, we only need to consider that the way to change (update) the mental state of an agent is by performing actions. However, we will instantiate such generic $ms_i$ by either a BU_{Unity} or a BU_{UpL} mental state whenever the distinction is necessary.

In order to control the behaviour of a multi-agent system we introduce action-based choreographies. We understand them as protocols which dictate the way agents behave by imposing ordering and synchrony constraints on their action executions. They represent exogenous coordination patterns and they can be seen as an alternative to message passing communication, with the potential advantage of not needing to establish a “common communication language”. Choreographies are useful in scenarios where action synchrony is more important than data.
2.4.1 Action-based Choreographies

For the ease of presentation, we represent choreographies as regular expressions where the basic elements are pairs \((i, a)\). Such pairs denote that the agent \(i\) performs the action \(a\). They can be combined by sequence, parallel, choice or Kleene operators, with the usual meaning: \((i_1, a_1);(i_2, a_2)\) models orderings, agent \(i_1\) executes \(a_1\) which is followed by agent \(i_2\) executing \(a_2\); \((i_1, a_1) || (i_2, a_2)\) models synchronisations between actions, agent \(i_1\) executes \(a_1\) while \(i_2\) executes \(a_2\); \((i_1, a_1) + (i_2, a_2)\) models non-deterministic choices, either \(i_1\) executes \(a_1\) or \(i_2\) executes \(a_2\); \((i_1, a_1)\)\(^*\) models iterated execution of \(a\) by \(i\). The operators respect the usual precedence relation\(^2\). The BNF grammar defining a choreography is as follows:

\[
c ::= (id, a) \mid c + c \mid c \parallel c \mid c; c \mid c^*
\]

In order to describe the transitions of a multi-agent system in the presence of a choreography \(c\), we first associate a transition system \(S^c\) to the choreography. We do this in the usual way, inductively on the size of the choreography such that the labels are of the form \(\parallel_{i \in I} (i, a_i)\). Such a transition system always exists (see [239] or [90] for a direct deterministic construction using the derivatives of a given regular expression).

We take, for instance, the transition system from Figure 2.5 which is associated with the choreography \(c\) defined as the following regular expression:

\[
c = (i_1, clean) || (i_2, move(C,A, floor));
(i_1, move(B, floor,A)); ((i_1, move(B, floor,A)) || (i_2, clean)) +
(i_2, move(B, floor,A)); ((i_2, move(B, floor,A)) || (i_1, clean)).
\]

The choreography specifies that two agents \(i_1, i_2\) work together in order to build the tower \(ABC\) and that furthermore, while one is building the tower the other one is cleaning the floor. More precisely, the definition of the choreography says that first \(i_2\) deconstructs the initial tower (by moving the block \(C\) on \(floor\)) while \(i_1\) is synchronously cleaning; next, either \(i_1\) constructs the final tower while \(i_2\) cleans or the other way around; afterwards, the system is in a final state. Further variations (like for example, in the case of a higher tower, one agent builds an intermediate shorter tower leaving the other to finish the construction) are left to the imagination of the reader.

We denote by \(S^c \otimes I\) the synchronised product of a choreography \(c\) and a multi-agent system \(I\). The states of \(S^c \otimes I\) are pairs \((cs, M)\) where \(cs\) is a state of \(S^c\) and \(M\) is a state of the multi-agent system \(I\). The transition rule for \(S^c \otimes I\) is given as follows:

\[
\frac{cs \rightarrow a_j^{a_j}}{\bigwedge_{j \in J} ms_j^{a_j} \rightarrow ms_j'} (mas)
\]

\[
(c, M) \rightarrow (c', M')
\]

\(^2\) If we denote \(\preceq^p\) the precedence relation, then we have \('+' \preceq^p '||' \preceq^p ';;' \preceq^p '\ast'\)
where $cs, cs'$ are states of $S^c$, $l$ is a choreography label of the form $\parallel_{j \in \mathcal{J}} (j, a_j)$ with $\mathcal{J}$ being a subset of $I$, $ms_j, ms'_j$ are mental states of agent $j$ and $M, M'$ are states of the multi-agent system with $M'$ being $M \setminus \{(j, ms_j) \mid j \in \mathcal{J}\} \cup \{(j, ms'_j) \mid j \in \mathcal{J}\}$. The notation $ms_j \xrightarrow{a_j} ms'_j$ is used to denote that agent $j$ performs action $a_j$ (eventually with $\tau$ steps) in $ms_j$ resulting in $ms'_j$. “Eventually $\tau$ steps” is needed for agents performing internal actions, like making choices among plans or handling failures in the case of BUpL agents. In the case of agents “in the style of BUnity”, $\xrightarrow{a}$ is simply $\rightarrow$ since Bunity agents do not have $\tau$ steps.

The transition rule $(mas)$ says that the multi-agent system advances one step when the agents identified by $\mathcal{J}$ perform the actions specified by the label $l$. The new state of the multi-agent system reflects the updates of the mental states of the individual agents.

### 2.4.2 A Finer Notion of Refinement

We would like to have the result that if the agents (for example BUpL) in a multi-agent system $I_1$ are refining the (BUnity) agents in $I_2$ then $S^c \otimes I_1$ is a refinement of $S^c \otimes I_2$. When refinement is defined as trace inclusion, this is, indeed, the case, as we can shortly prove in Proposition 2.2.

**Proposition 2.2.** Given two multi-agent systems $I_1, I_2$ such that $(\forall i_1 \in I_1)(\exists i_2 \in I_2) (ms_{i_1} \subseteq ms_{i_2})$ and a choreography as $S^c$ we have that $S^c \otimes I_1 \subseteq S^c \otimes I_2$.

**Proof.** Let $M_1$ and $M_2$ be the initial states of the multi-agent systems $I_1$ and $I_2$. Let also $cs_0$ be the initial state of the transition system $S^c$ associated to the choreography $c$. It is enough to notice that $Tr((cs_0, M_1)) = Tr(cs_0) \cap Tr(M_1)$ and that $ms_{i_1} \subseteq ms_{i_2}$ for all $i_1 \in I_1$ implies $Tr(M_1) \subseteq Tr(M_2)$.

However, adding choreographies to a multi-agent system may introduce deadlocks. On the one hand, we would like to be able to infer from the semantics when
a multi-agent system is in a deadlock state. On the other hand, we would like to have that the refinement of multi-agent systems does not introduce deadlocks. Trace semantics is a too coarse notion with respect to deadlocks. There are two consequences: neither is it enough to define the semantics of a multi-agent system as the set of all possible traces, nor is it satisfactory to define agent refinement as trace inclusion. We further illustrate these affirmations by means of simple examples.

We take, for instance, the choreography \( c = (i, \text{move}(B, floor, C)) \), where \( i \) symbolically points to the BUuL agent from Section 2.2.5. Looking at the plans and repair rules of the BUuL agent we see that such an action cannot take place. Thus, conforming to the transition rule \( (\text{mas}) \), there is no possible transition for the product \( S^c \otimes I \). Just by analysing the behaviour (the empty trace) we cannot infer anything about deadlocked states: is it that the agent has no plan, or is it that the choreography asks for an impossible execution? This is the reason why, in order to distinguish between successful and deadlocked executions, we explicitly define a transition label \( \sqrt{\text{diff}} \) different from any other action relations. We then define for the product \( S^c \otimes I \) an operational semantics \( O^\sqrt{\text{diff}}(S^c \otimes I) \) as the set of maximal (in the sense that no further transition is possible) traces, ending with \( \sqrt{\text{when the last state is successful:} \}

\[
\{ tr \sqrt{\text{diff}} | (cs_0, M_0) \xrightarrow{tr} (cs, M) \Rightarrow cs \in F(S^c) \} \cup \\
\{ tr | (cs_0, M_0) \xrightarrow{tr} (cs, M) \Rightarrow, cs \notin F(S^c) \} \cup \{ \epsilon | (cs_0, M_0) \Rightarrow \},
\]

where \( tr \) is a trace with respect to the transition rule \( (\text{mas}) \), \( M_0 \) (resp. \( cs_0 \)) is the initial state of \( I \) (resp. \( S^c \)) and \( \epsilon \) denotes that there are no possible transitions from the initial state.

We can now define the refinement of multi-agent systems with respect to the above definition of the operational semantics \( O \).

**Definition 2.11 (MAS Refinement).** Given a choreography \( c \), we say that two multi-agent systems \( I_1 \) and \( I_2 \) are in a refinement relation if and only if the set of maximal traces of \( S^c \otimes I_1 \) are included in the set of maximal traces of \( S^c \otimes I_2 \). That is, \( O^\sqrt{\text{diff}}(S^c \otimes I_1) \subseteq O^\sqrt{\text{diff}}(S^c \otimes I_2) \).

We now approach the problem that appears when considering agent refinement defined as trace inclusion. It can be the case that the agents in the concrete system refine (with respect to trace inclusion) the agents in the abstract system, nevertheless the concrete system deadlocks for a particular choreography. We take, for instance, the BUuL and BUuL agents from Sections 2.2.3 and 2.2.5. For the ease of reference, we identify the BUuL agent by \( i_a \) (since it is more abstract) and the BUuL agent by \( i_c \) (since it is more concrete). We can easily design a choreography which works fine with \( i_a \) (does not deadlock) and on the contrary, it deadlocks with \( i_c \). Such a choreography is for example the one mentioned in the beginning of the section, \( c = (i, \text{move}(B, floor, C)) \), where, \( i \) points now to either \( i_a \) or \( i_c \) up to a renaming. We recall that \( i_c \) is a refinement of \( i_a \). However, we have already mentioned, \( i_c \) cannot execute the move (since the move is irrelevant for building the ABC tower and at
implementation time it matters to be as precise as possible), while \( i_a \) can (since in a specification “necessary” is more important than “sufficiency”).

What the above illustration implies is that refinement as trace inclusion, though being a satisfactory definition at individual agent level, is not a strong enough condition to ensure refinement at a multi-agent level, in the presence of an arbitrary choreography. It follows that we need to redefine individual agent refinement such that multi-agent system refinement (as maximal trace inclusion) is compositional with respect to any choreography. In this sense, a choreography is more like a context for multi-agent systems, meaning that whatever the context is, it should not affect the visible results of the agents’ executions but restrict them by activating only certain traces (the other traces still exist, however, they are inactive).

In order to have a proper definition of agent refinement we look for a finer notion of traces. The key ingredient lies in enabling conditions for actions. Given a mental state \( ms \), we look at all the actions enabled to be executed from \( ms \). We denote them by \( E(ms) = \{a \in A \mid \exists ms'(ms \xrightarrow{a} ms')\} \) and we call \( E(ms) \) a ready set. We can now present ready traces as possibly infinite sequences \( X_1, a_1, X_2, a_2, \ldots \) where \( ms_0 \xrightarrow{a_1} ms_1 \xrightarrow{a_2} ms_2 \ldots \) and \( X_{i+1} = E(ms_i) \). We denote the set of all ready traces from a state \( ms_0 \) as \( RT(ms_0) \). Compared to the definition of traces, ready traces are a much more finer notion in the sense that they record not only actions which have been executed but also sets of actions which are enabled to be executed at each step.

**Definition 2.12 (Ready Agent Refinement).** We say that two agents with initial mental states \( ms \) and \( ms' \) are in a ready refinement relation (i.e., \( ms \subseteq rt ms' \)) if and only if the ready traces of \( ms \) are included in the ready traces of \( ms' \) (i.e., \( RT(ms) \subseteq RT(ms') \)).

We can now present our main result which states that refinement is compositional, in the sense that if there is a ready refinement between the agents composing two multi-agent systems it is then the case that one multi-agent system refines the other in the presence of any choreography.

**Theorem 2.2.** Let \( I_1, I_2 \) be two multi-agent systems such that \((\forall i_1 \in I_1) (\exists i_2 \in I_2) (ms_{i_1} \subseteq rt ms_{i_2})\) and a choreography \( c \) with the associated LTS \( S_c \). We have that \( I_1 \) refines \( I_2 \), that is, \( O^\forall(S_c \otimes I_1) \subseteq O^\forall(S_c \otimes I_2) \).

**Proof.** What we need to further prove with respect to Proposition 2.2 is that the set of enabled actions is a key factor in identifying failures in both implementation and specification. Assume a maximal trace \( tr \) in \( O^\forall(S_c \otimes I_1) \) leading to a non final choreography state \( cs \). Given \( cs_0 \) and \( M_1 \) as the initial states of \( S_c \), \( I_1 \), we have that \((cs_0, M_1) \xrightarrow{tr} (cs, M) (cs, M) \xrightarrow{l} \) for all \( l = I_{j \in J} (j, a_j) \) such that \( cs \xrightarrow{l} cs' \). By rule (mas) this implies that there exists an agent identified by \( j \) which cannot perform the action indicated. Thus the corresponding trace of \( j \) ends with a ready set \( X \) with the property that \( a_j \) is not included in it. We know that each implementation agent has a corresponding specification, be it \( j' \), such that \( j \) ready refines \( j' \). If we, on the
other hand, assume that \( j' \) can, on the contrary, execute \( a_j \) we would have that in a given state \( j' \) has besides the ready set \( X \) another ready set \( Y \) which includes \( a_j \). This contradicts the maximality of the ready set. \( \square \)

As a direct consequence of the above theorem, we are able to infer the absence of deadlock in the concrete system from the absence of deadlock in the abstract one:

**Corollary 2.1.** Let \( I_1, I_2 \) be two multi-agent systems with initial states \( M_1 \) and \( M_2 \). Let \( c \) be a choreography with the associated LTS \( S^c \) and initial state \( cs_0 \). We have that if \( I_1 \) refines \( I_2 \) \((\mathcal{O}^c(S^c \otimes I_1) \subseteq \mathcal{O}^c(S^c' \otimes I_2)\) and \( c \) does not deadlock the specification \((cs_0, M_2) \models □\neg⊥\) it is then also the case that \( c \) does not deadlock the implementation \((cs_0, M_1) \models □\neg⊥\).

As we have already explained in Section 2.3, proving refinement by deciding trace inclusion is an inefficient procedure. This is also the case with ready refinement, thus a more adequate approach is needed. If previously we have adopted simulation as a proof technique for refinement, now we consider weak ready simulation.

**Definition 2.13 (Weak Ready Simulation).** We say that two agents with initial mental states \( ms \) and \( ms' \) are in a (weak) ready simulation relation \((ms ≲_{rs} ms')\) if and only if \( ms ≲ ms' \) and the corresponding ready sets are equal \((E(ms) = E(ms'))\).

As it is the case for simulation being a sound and complete proof technique for refinement, analogously we can have a similar result for ready simulation. We recall that determinacy plays an important role in the proof for completeness.

**Proposition 2.3.** Given two agents with initial mental states \( ms \) and \( ms' \), where the one with \( ms \) is deterministic, we have that \( ms ≲_{rs} ms' \) iff \( ms' \subseteq rt ms \).

**Remark 2.2.** For the sake of generality, in the definitions from this section we have used the symbolic notations \( ms, ms' \). BUnity and BUpL agents can be seen as (are, in fact) instantiations. Proposition 2.3 relates to Proposition 2.1. The only difference is that, for simplification, Proposition 2.3 refers directly to ready simulation and not to its modal characterisation, as it was the case for simulation in Proposition 2.1. It is not difficult to adapt Definition 2.9 to the ready simulation. One needs only to change the condition on the transition \((mas)\) from \((B, p) \xrightarrow{a} (B', p')\) to the conjunction \((B, p) \xrightarrow{a} (B', p') \land E((B, p)) = E((B))\) which checks also the equality on the ready sets.

Recalling the BUpL and BUnity agents \( i_c \) and \( i_a \), we note that though \( i_a \) simulates \( i_c \) it is not also the case that it ready simulates. This is because the ready set of the BUnity agent is always larger than the one of the BUpL agent. One basic argument is that \( i_a \) can always “undo a block move”, while \( i_a \) cannot. However, let us see what would have happened if we were to consider changing \( i_a \) by replacing the conditional action set from Figure 2.1 with the set from Figure 2.6:
We now have a BUnity agent which is less abstract. Basically, the instantiation from the first two conditional actions disallows any spurious “to and fro” sequence of moves like \( \text{move}(X,Y,Z) \) followed by \( \text{move}(X,Z,Y) \) which practically undoes the previous step leading to exactly the previous configuration. The instantiation is obvious when one looks at the final “desired” configuration. The last conditional action allows “destructing” steps by moving blocks on the floor. It can still be considered as a specification. It provides no information about the order of executing the moves since this is not important at the abstraction level. With the above change, the new BUnity agent ready simulates \( i_c \). To see this, it suffices to notice that the only \( \text{BUpl} \) ready trace is \( \{ \text{move}(C,A,\text{floor}) \} \), \( \text{move}(C,A,\text{floor}) \), \( \{ \text{move}(B,\text{floor},A) \} \), \( \text{move}(B,\text{floor},A) \), \( \{ \text{move}(C,\text{floor},B) \} \), \( \text{move}(C,\text{floor},B) \) which is also the only BUnity ready trace. The same equality of ready traces holds when we consider the additional \text{clean} \) action.

We recall the choreography from Figure 2.5 and we consider a BUnity multi-agent system which consists of two copies of \( i_a \) (enabled to execute also \text{clean}). For either branch, the executions (with respect to the transition \( \text{(mas)} \)) of the multi-agent system are successful (the choreography reaches a final state). Since \( i_c \) ready refines \( i_a \), by Corollary 2.1 we can deduce that also the executions of a multi-agent system which consists of two \( i_c \) copies are successful.

### 2.5 Timing Extensions of MAS

Modelling time in multi-agent systems make them more expressive. For example, timing constraints can be used to enforce delays between actions and to time restrict action execution, that is, to force action execution to happen before certain time invariants are violated. Our approach in adding time to multi-agent systems consists of adapting the theory of timed-automata [10]. A \textit{timed automaton} is a finite transition system extended with real-valued clock variables. Time advances only in states since transitions are instantaneous. Clocks can be reset at zero simultaneously with any transition. At any instant, the reading of a clock equals the time elapsed since the last time it was reset. States and transitions have \textit{clock constraints}, defined as:

\[
\phi_c ::= x \leq t \mid t \leq x \mid x < t \mid t < x \mid \phi_c \land \phi_c,
\]

where \( t \in \mathbb{Q} \) is a constant and \( x \) is a clock. When a clock constraint is associated with a state, it is called \textit{invariant}, and it expresses that time can elapse in the state as long as the invariant stays true. When a clock constraint is associated with a transition,
it is called guard, and it expresses that the action may be taken only if the current values of the clocks satisfy the guard.

In our multi-agent setting, timed choreographies are meant to impose time constraints on the actions executed by the agents. We model them as timed automata. We take, as an example, the choreography from Figure 2.7. There is a single clock $x$. The initial state $cs_0$ has no invariant constraint and this means that an arbitrary amount of time can elapse in $cs_0$. The clock $x$ is always reset with the transition from $cs_0$ to $cs_1$. The invariant $x < 5$ associated with the state $cs_1$ ensures that the synchronous actions clean and move($C,A,floor$) must be executed within 5 units of time. The guard $x > 6$ associated with the transition from $cs_2$ to $cs_3$ ensures that the agents cannot spend an indefinite time in $cs_2$ because they must finish their tasks after 6 units of time.

$$\begin{align*}
\text{cs}_0 & \xrightarrow{(i_2,\text{move}(C,A,floor))} \text{cs}_1 \\
& \xrightarrow{(i_1,\text{clean}),x:=0} \\
& \xrightarrow{(i_1,\text{move}(B,floor,A))} \text{cs}_2 \\
& \xrightarrow{(i_1,\text{move}(C,floor,B)),x>6} \text{cs}_3
\end{align*}$$

Fig. 2.7 A timed choreography

We now approach the issue of modelling time in BUnity and BUpL agents. In this regard, we consider that agents have a set of local clocks and that clock valuations can be performed by an observer. We further pose the problem of how agents make use of clocks. We recall the design principle: “the specification of basic actions does not come with time”, thus actions are instantaneous. This implies that, in order to make the time pass, we need to extend the syntax of the agent languages with new application specific constructions such that the ontology of basic actions remains timeless (basic actions being specified only in terms of pre/post conditions). This is why we introduce delay actions, $\phi \rightarrow I$, where $\phi$ is a query on the belief base and $I$ is an invariant like $x \leq 1$. Basically, their purpose is to make time elapse in a mental state where certain beliefs hold. As long as the invariant is true, the agent can stay in the same state while time passes. We refer to $\mathcal{D}$ as the set of delays of either a BUnity or a BUpL agent. This is because, as it is the case for basic actions, delays are syntactical constructions belonging to both BUpL and BUnity languages. In what follows, we discuss the time extension for each language separately.

### 2.5.1 Adding Time to BUnity

We now focus on time extending BUnity conditional actions. First, the queries of conditional actions are defined both on belief bases and clock valuations. Second, conditional actions specify the set of clocks to be reset after the execution of basic actions. Their syntax becomes $\{\phi \wedge \phi_c\} \triangleright do(a), \lambda$. Timed conditional actions are
meant to say that if certain beliefs $\phi$ hold in the current mental state of a BUnity agent (as before) and additionally, certain clock constraints $\phi_c$ are satisfied, then the basic action $a$ is executed and the clocks from the set $\lambda$ are reset to 0. Taking into account the previous discussion of the mechanism of delay actions, the corresponding changes in the semantics are reflected in Figure 2.8: where $\lambda$ is the set of clocks reset by performing action $a$ and $\nu$ represents the current clock valuations. We use the notation $\nu \in I$ (resp. $\nu \in \phi_c$) to say that the clock valuations from $\nu$ satisfy the invariant $I$ (resp. the constraint $\phi_c$). When $\phi_c$ is absent we consider that trivially $\nu \in \phi_c$ holds.

We make a short note that our design decision is to separate the implementation of delays from the one of conditional actions. This is because a construction like $\{\phi\} \triangleright I, do(a), \lambda \trianglerightdo(a)$, $\lambda$ is ambiguous. If $\phi$ holds, it can either be the case that time elapses with respect to the invariant $I$ and $a$ is suspended, or that $a$ is immediately executed. However, it sometimes is important to ensure that “time passes in a state”, instead of leaving this only as a non deterministic choice.

To illustrate the above constructions we recall the BUnity agent $i_a$ from Figure 2.6. We basically extend the BUnity agent such that the agent has one clock, be it $y$, which is reset by conditional actions, and such that the agent can delay in given states, thus letting the time pass.

$$C = \{ \top \triangleright (do(clean), y := 0), \neg on(B, A) \triangleright do(move(B, floor, A)), \neg on(C, B) \land on(B, A) \triangleright do(move(C, floor, B)), \neg (on(B, A) \land on(C, B)) \triangleright (do(move(X, Y, floor)), y := 0) \}$$

$$D = \{ on(C, floor) \lor cleaned \leftarrow (y < 9), on(B, A) \lor cleaned \leftarrow (y < 10) \}$$

Figure 2.9 Extending $i_a$ with clock constraints

Figure 2.9 shows a possible timed extension. The clock $y$ is reset after either performing clean or moving a block on the floor. The agent can delay until the clock valuates to 9 (resp. 10) units of time after moving $C$ on the floor (resp. $B$ on $A$).
2.5.2 Adding Time to BUpL

The timed extension of BUpL concerns changing plans such that previous calls \( a;p \) are replaced by \((\phi_c,a,\lambda);p\) and \((\phi \rightarrow I);p\), where \( \phi_c \) is time constraining the execution of action \( a \) and \( \lambda \) is the set of clocks to be reset. To simplify notation, if clock constraints and clock resets are absent we use \( a \) instead of \((a)\).

We make the remark that if previously actions failed when certain beliefs did not hold in a given mental state, it is now the case that actions fail also when certain clock constraints are not satisfied. Consider, for example, the plan \((\langle x < 1\rangle, a,[x := 0]);(x > 2),b,\emptyset)\). There is no delay action between \( a \) and \( b \), thus the time does not pass and \( x \) remains 0, meaning that \( b \) cannot be executed. Such situations are handled by means of the general semantics of the repair rules. There are two possibilities: either to execute an action with a time constraint that holds, or to make time elapse. The latter is achieved by triggering a repair rule like \( true \leftarrow \delta \), where for example \( \delta \) is a delay action \( true \rightarrow true \) which allows an indefinite amount of time to pass. The corresponding changes in the semantics are reflected in Figure 2.10:

![Transition Rules for Timed BUpL](image)

To see a concrete example, we recall the BUpL agent from Section 2.2.6. We consider two delay actions \( true \rightarrow (y < 9) \) and \( true \rightarrow (y < 10) \). We further make the delays and the clock resets transparent in the plans. The plan \( cleanR \) changes to \( true \rightarrow (\langle y < 9\rangle);\) cleanR such that the clock \( y \) is reset after a clean action. The plan \( rearrange(x_1,x_2,x_3) \) changes to \( true \rightarrow (\langle y < 9\rangle);\) move\((x_1, floor, x_2)\); \( true \rightarrow (\langle y < 10\rangle);\) move\((x_3, floor, x_1)\) such that time passes between moves.

The observable behaviour of either timed BUnity or BUpL agents is defined in terms of timed traces. A **timed trace** is a (possibly infinite) sequence \((t_1,a_1),(t_2,a_2)\ldots(t_i,a_i)\ldots\) where \( t_i \in \mathbb{R}_+ \) with \( t_i \leq t_{i+1} \) for all \( i \geq 1 \). We call \( t_i \) a **time-stamp** of action \( a_i \) since it denotes the absolute time that passed before \( a_i \) was executed. We then have that a timed BUnity or BUpL agent computation over a timed trace \((t_1,a_1)(t_2,a_2)\ldots(t_i,a_i)\ldots\) is a sequence of transitions:

\[
ms_0,v_0 \xrightarrow{\delta_1} a_1 ms_1,v_1 \xrightarrow{\delta_2} a_2 ms_2,v_2 \ldots
\]

where \( ms_i \) is a BUnity (BUpL) mental state and \( t_i \) are satisfying the condition \( t_i = t_{i-1} + \delta_i \) for all \( i \geq 1 \).
For example, a possible timed trace for either the timed BUpl or BUnity agent is 
\((0, \text{clean}),(7, \text{move}(C,A,\text{floor})), (8, \text{move}(B,\text{floor},A)), (9, \text{move}(C,\text{floor},B))\). It is, in fact, the case that any BUpl timed trace is also a BUnity timed trace, thus the two agents are again in a refinement relation. We elaborate more on timed refinement in the next section.

2.5.3 A Short Note on Timed Refinement

We recall that a key element in having simulation as a proof technique for individual agent refinement was the determinacy of BUnity agents. We note that in order to have a similar “timed” result we only need to impose the restriction that clock constraints associated with the same action must be disjoint. This ensures determinacy of timed automata. A weaker restriction (which nevertheless requires a “determinisation” construction) is to require that each basic action is associated with at most one clock and that conditional actions can only reset the clock corresponding to the basic action being executed; however, the guards in conditional actions may consult different clocks. Under the disjointness condition, we have that timed BUnity agents are deterministic, thus the same proof technique as in Section 2.3 can be applied. We make the remark that timed simulation differs from simulation in only one aspect: we further need to consider simulating \(\delta\) steps and not only \(a\) steps.

As for multi-agent system refinement, we recall that in Section 2.4.1 we considered that agents are associated with identifiers. We now need to consider that agents have also a set of clocks to manipulate. We thus update the definition of a multi-agent state \(M\) as being \(\{(i,ms_i,\nu_i) \mid i \in I\}\), where \(I\) is the set of agent identifiers, \(ms_i\) is a mental state for the agent \(i\) and \(\nu_i\) represents the clock valuations of \(i\).

We further recall that we gave semantics to multi-agent systems by means of a transition rule for the synchronised product of the system and a given choreography. Adding time constructions to both agent languages and to the choreography needs to be reflected by changing the transition rule \((\text{mas})\) correspondingly. More precisely, we first need a transition for passing time which corresponds to delays in both choreography and agent states. We then need to change the rule \((\text{mas})\) such that a transition of the system is enabled not only if certain agents can perform the actions specified by the choreography, but also if the clock valuations of the acting agents satisfy the clock constraints of the choreography. The changes are illustrated in Figure 2.11, where \(l\) is \(\|_{j \in J} (j,a_j)\), \(g\) is the clock constraint associated with the label \(l\), \(I(cs)\) denotes the invariant associated with the state \(cs\) and \(M'\) is \(M \setminus \{(j,ms_j,\nu_j) \mid j \in J\} \cup \{(j,ms'_j,\nu'_j) \mid j \in J\}\).

We note that the transition \((\text{delay})\) can take place only if all agents are able to delay. No deadlocks are introduced since delays are not compulsory.

Similarly as in Section 2.4.1 the semantics of a timed agent system together with a timed choreography is defined as the set of maximal timed computations where
we make the distinction between a success and a deadlock. This was needed (and it still is) in order to reason about deadlock. We recall that we further needed to change refinement (resp. simulation) to ready refinement (resp. ready simulation). The reason was that in order to have compositionality of multi-agent systems refinement one needs to make sure that any choreography which does not deadlock the abstract system cannot deadlock the concrete one. The same reasoning applies in the case of timed agent systems. We need to investigate if further deadlock situations can arise which are not taken into account in the framework of ready refinement.

As we have already mentioned, the transition \((\text{delay})\) does not introduce deadlock situations. Thus, we only need to consider the transition \((\text{t-mas})\). What is new with respect to the previous transition \((\text{mas})\) is the additional condition which asks that the clock valuations satisfy the guard of the current transition in the choreography. We note that the fact that a clock constraint in the implementation does not satisfy the guard of the choreography at a given time is visible at the level of the timed ready sets. Thus, the timed version of ready refinement suffices in order to have compositionality of timed agent systems refinement. Baring this in mind, we only need to focus on timed ready simulation as being a proof technique for timed agent systems refinement, which previously was a valid statement as a consequence of the determinacy of the choreography. Thus, along the same line as before, we only need to require that clock constraints associated with the same action are disjoint such that timed choreographies are deterministic.

As an observation, a timed BUpL multi-agent system consisting of two instances of the agent described in Section 2.5.2 running under the timed choreography from Figure 2.7 is a timed refinement of a timed BUnity multi-agent system consisting of two instances of the agent described in Section 2.5.1 running under the same choreography. Furthermore, both systems are deadlock free. To illustrate this latter affirmation, we present a small experiment in UPPAAL [31], a tool for verifying timed automata. At a more abstract and syntactic level, we have modelled the timed choreography and the timed BUpL agent as timed automata in UPPAAL. We have then verified that the value of the clocks are always greater than 6. This implies that the choreography always reaches the final state. Figure 2.12 illustrates the timed BUpL system consisting of two instances of the BUpL agent and the choreography. The BUpL agent is parametrised by \(id_b\), a bounded integer variable which is in our case 0 or 1. We note that we had to “approximate” and implement the parallel

---

**Fig. 2.11** Transition Rules for a Timed Agent System
operator using an interleaving mechanism (first one agent cleans and after the other one moves \(C\) on the floor). The synchronisation between the choreography and the BUpL agents is in the CCS style (e.g., clean[1-e]! and clean[1-e]?).

![Diagram of a Timed BUpL System Modelled in UPPAAL](image)

Using the UPPAAL Simulate command one can experiment with different timed executions of the system. Figure 2.13 represents one of them. The trace shows that the BUpL instance \(Bp(0)\) is the first to execute clean followed by \(Bp(1)\) executing the destructing step (\(C\) on the floor). From this point \(Bp(0)\) finishes the \(ABC\) tower. Finally, \(Bp(1)\) executes, at its turn, the action clean.

### 2.6 Conclusion

We have addressed the problem of multi-agent system refinement. We have first focused on individual agent refinement where we relied on fair simulation as a proof technique for fair trace inclusion. We have then extended the notion of refinement of individual agents to multi-agent systems, where the behaviour of the agents composing the systems is coordinated by choreographies. Our approach to introducing choreographies to multi-agent systems consisted of defining them as action-based coordination mechanisms. In such a framework, we have the results that agent refinement is a sufficient condition for multi-agent system refinement and that this latter notion preserves deadlock freeness. We have further illustrated a timed extension of multi-agent systems by means of timed automata where the same refinement methodology can be adapted.
We have stressed the importance of verification from the introduction. Our goal was to describe a general methodology for a top-down design of multi-agent systems which makes it simple to execute and verify agent programs. Concerning this practical side we mention that we have already implemented our formalism in Maude. Maude is an encompassing framework where we prototyped the agent languages we described such that it is possible to (1) execute agents by rewriting; (2) verify agents by means of simulation, model-checking, searching, or testing. Since we were mainly interested in refinement, the properties we focused on were correctness properties, i.e., model-checking for the absence of deadlock in the product of a BUpL and BUnity agent. However, we have experimented with different other safety and liveness properties. In this regard, please see Chapter [421] for further details and references. The current version of the implementation (also including the timed languages prototyped in Real-Time Maude [326]) can be found at http://homepages.cwi.nl/~astefano/agents. Further extensions with respect to model-checking timed agents and automatically generating test cases for verifying infinite state agents need to be investigated.
Specification and Verification of Multi-agent Systems
Dastani, M.; Hindriks, K.V.; Meyer, J.-J. (Eds.)
2010, XVII, 405 p., Hardcover