On Relating Functional Specifications to Architectural Specifications: A Case Study

Flavio Corradini,[†] Paola Inverardi,[†] and Alexander L. Wolf[‡]

[†]Dept. di Informatica
Universita' dell' Aquila
I-67010 L'Aquila, Italy
{flavio,inverard}@univaq.it

[‡]Dept. of Computer Science
University of Colorado
Boulder, CO 80309-0430 USA
alw@cs.colorado.edu

University of Colorado
Department of Computer Science
Technical Report CU-CS-933-02 June 2002

© 2002 Flavio Corradini, Paola Inverardi, and Alexander L. Wolf

ABSTRACT

Software architecture specifications are predominantly concerned with describing the component structure of systems and how the components interact behaviorally. Little attention has been paid to formally relating those specifications to higher levels of specification, such as the system requirements. In this paper we present our progress toward addressing an instance of this problem, namely relating state-based software architecture specifications to high-level functional specifications. Our approach is to use an intermediate specification given in terms of a set of temporal logic properties to bridge the gap between the two levels of specifications. We describe our approach in the context of a particular case study, the AEGIS GeoServer Simulation Testbed system, showing how a compact functional specification of a critical behavioral property of the system can be used to validate three alternative architectural specifications of that system.

1 Introduction

A software architecture specification describes the high-level design of a system in terms of the structural and communication relationships of its components. At this level of description, the specification focuses on the interaction behaviors exhibited by the components, not the algorithms used internally by components to carry out their functional roles.

It is believed that having such design specifications for complex systems can improve the effectiveness of the development process and the quality of the final product by permitting analyses to be performed early in the life cycle [5, 13, 23]. Consequently, much effort has been devoted to researching, experimenting with, and categorizing the kinds of analyses that can be performed and the product quality attributes that can be improved through them [1, 3, 4, 7, 8, 9, 14, 15, 17, 18, 19, 21, 22].

Despite the significant amount of research in architectural specification, there has been little attempt to formally relate it to higher levels of specification, such as the system requirements [24]. Only recently has interest begun to emerge in this regard [16, 20]. In this paper we present our progress toward addressing an instance of this problem, namely relating high-level functional specifications to software architecture specifications. We describe our approach in the context of a particular case study, the AEGIS GeoServer Simulation Testbed (AGST) system [2, 12], showing how a compact functional specification of a critical behavioral property of the system can be used to validate three alternative architectural specifications of that system.

The basis for the method is the construction of a set of temporal logic properties that bridge the gap between the functional specification and the architectural specification. In essence, the temporal logic properties make explicit and visible the relevant behavioral dependencies that are inherent in the functional specification and that are deeply buried within the complex state descriptions of the architectural specification. Of course, both the functional and architectural specifications are abstractions, and as such they embody simplifying assumptions. These assumptions appear both at the level of individual components (i.e., so-called "black-box" assumptions) and at the level of the global architecture. The process of relating the functional and architectural specifications helps the analyst understand the interplay of the assumptions operating at these two levels.

The temporal logic properties developed here are given using the CTL formalism [10, 11]. This is a well-known branching logic for which model checkers are available. To specify software architectures, we use the CHAM (CHemical Abstract Machine) formalism [6]. This formalism has been used extensively by us and others in previous work on architectural specification and analysis [9, 14, 15, 25]. Note, however, that the general approach we describe in this paper is not tied to the CHAM formalism. For those unfamiliar with CHAM or its use in specifying software architectures, Appendix A provides a brief review. Appendix A also provides a brief review of CTL.

In the next section of the paper we introduce the subject of our case study, AGST. We also provide the functional specification that captures the property of concern. In Section 3 we give the CHAM specifications of three architectural variants of AGST. The CTL specifications that relate the architectural variants to the functional specification are presented in Section 4. We conclude in Section 5 with a discussion of future work.

2 The AEGIS GeoServer Simulation Testbed

The AEGIS GeoServer Simulation Testbed (AGST) was presented as a challenge problem for software architecture research. The problem was first described by Balzer [12] and later investigated by Allen and Garlan [2].

The problem derives from the U.S. Navy's AEGIS radar and missile defense system. AEGIS is

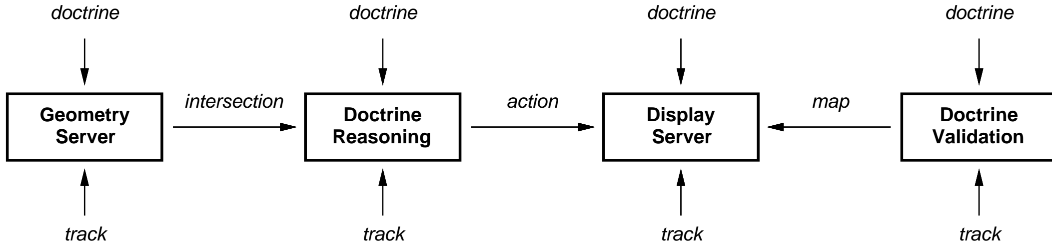

Figure 1: Data Flow in AGST.

a shipboard semi-automatic weapon designed to monitor potential threats, such as hostile aircraft movements, to propose an appropriate response, such as firing a missile, and to carry out that response. The purpose of the next-generation GeoServer system is to combine data from several AEGIS-equipped ships in order to better determine which targets are in which regions, how those targets are moving among regions, and which ship will provide a response.

AGST consists of four primary functional elements,[1] as depicted in the data-flow diagram of Figure 1. The primary parameters to these computations are *doctrines*, which are conditions used to classify a threat as real (e.g., an unidentified moving object descending rapidly toward an aircraft carrier), and *tracks* of moving objects. Geometry Server (GS) computes the *intersections* between tracks and (stored) geometric regions by making use of doctrines. Doctrine Reasoning (DR) uses the intersections, doctrines, and tracks to compute *actions* to be carried out as responses to threats. Doctrine Validation (DV), given a doctrine and a track, computes *maps* of relevant regions. Finally, Display Server (DS) is responsible for dynamic graphical depictions of the simulated situation based on actions, maps, doctrines, and tracks.

A critical aspect of the AGST system, as described by Balzer, has to do with a particular relationship among the computations: It should be the case that the doctrine and track used to compute an action from an intersection should be the same pair used to compute the intersection itself. In turn, that same pair should be used to compute the associated map and the associated display. Thus, we have a global invariant that must be satisfied by any architecture correctly realizing the system.

2.1 Specifying the Requirement

To formally describe the invariant, we use a simple functional specification that clearly captures the relevant dependencies among the computations. Let us assume the following notation.

- $D = \{doctrine_1, \ldots, doctrine_m\}$, a set of doctrines.
- $T = \{track_1, \ldots, track_n\}$, a set of tracks.
- $I = \{intersection_1, \ldots, intersection_p\}$, a set of intersections.
- $A = \{action_1, \ldots, action_q\}$, a set of actions.

[1] Unfortunately, the names that we were given for the elements are not particularly consistent with the functional roles that those elements play within the system. Nevertheless, we retain these names to avoid confusion with any other published descriptions of the system.

- $M = \{map_1, \ldots, map_r\}$, a set of maps.
- $S = \{scene_1, \ldots, scene_t\}$, a set of displayed scenes.

We can now define the main computational functions, as follows.

- $\mathsf{GS} : D \times T \rightarrow I$
- $\mathsf{DR} : D \times T \times I \rightarrow A$
- $\mathsf{DV} : D \times T \rightarrow M$
- $\mathsf{DS} : D \times T \times A \times M \rightarrow S$

Then the correct behavior of the system with respect to the global invariant can be simply specified as follows.

$$\mathsf{DS}(\mathsf{DR}(\mathsf{GS}(doctrine_i, track_j), doctrine_i, track_j), \mathsf{DV}(doctrine_i, track_j), doctrine_i, track_j)$$

Clearly, this functional specification would form only one part of a larger specification that captured all the desired properties of the system relevant at this level of abstraction. Nevertheless, this functional specification is useful in analyzing the system. In fact, we can see that it embodies a significant amount of information.

One bit of information derives from the use of function application, which implicitly puts a partial ordering on the way data are processed. So, for example, the computation performed by GS must strictly precede that of DR, and DR before that of DS. But the computation performed by DV may occur before, during, or after DR. Moreover, the computations of DV and DR are logically independent.

Another bit of information derives from the use of explicit parameter naming, which allows one to keep track of the identity of data throughout the computation. In this case, within the context of a single computation performed by DS, the four computations represented by GS, DR, DV, and DS must all use the same instances, i and j, of doctrine and track data, respectively.

Considering together the partial order of computations and the constraint on the data used in those computations, the design problem we face is one of *coordination*. In particular, there are four distinct occurrences of doctrine/track pairs appearing in the expression of the functional requirements, and it becomes the responsibility of the designer to ensure, as noted above, that the global invariant on the data is maintained. Therefore, what we need in the design is a policy for guaranteeing the coherency of the data under these circumstances. Each of the architectures described in Section 3 embodies a different such coordination policy.

2.2 Bridging the Gap to Architectural Specification

The functional specification presented above is actually quite expressive. It gives information about the system behavior, as well as the relationships among the various system components. On the one hand, it assumes that the components behave as functions that take inputs and produce (unique) outputs, while on the other hand, through function application, the specification describes how the components are causally related, both sequentially (e.g., DR computes actions from intersections provided by GS) and concurrently (e.g., the outputs of DR and DV are computed independently).

Let us examine a simple portion of the functional specification in order to illustrate the steps needed to obtain the same amount of information in a state-based operational model. Consider $\mathsf{DR}(\mathsf{GS}(\mathit{doctrine}_i, \mathit{track}_j), \mathit{doctrine}_i, \mathit{track}_j)$. At a macro level, it expresses that DR and GS are causally related, since GS is supposed to provide an input to DR. Moreover, the notation $\mathsf{GS}(\mathit{doctrine}_i, \mathit{track}_j)$ in the context of the formula plays two distinct behavioral roles. For one it represents GS when it is getting the inputs, namely $\mathit{doctrine}_i$ and track_j. For another it stands for the produced output. In an operational context these two situations are clearly distinct, and are represented as the state in which GS is ready to take its inputs and the state in which GS has produced an output. In between these two states, in terms of the global system behavior, other global states can be observed.

In order to formally relate the functional and architectural specifications we need to identify those states that are relevant for validating the functional properties. We must also describe the way the states should be causally related in order to satisfy the properties. Finally, we must prove that, in all possible computations of the architectural model, the states are reached in an order that guarantees the constraints to be satisfied. What we describe in this paper is a method for carrying out this process.

The method is based on two levels of specification. The first level consists of a set of CTL temporal logic *scheme* formulae that allow the expression of functional properties in terms of the behavioral properties of the global system. In fact, in this operational setting, we are observing the global system states and the way they are obtained along a computation. Computations are the result of concurrent activities of the system components and of their explicit synchronization through suitable synchronization primitives. The second level is formed by instantiating the scheme formulae in terms of concrete state properties. The instantiated scheme formulae are assembled together, using conjunction and disjunction operators, to reflect the behavior of the architecture. Note that the first level is driven by the generic semantics of functional specification and, therefore, is reusable. The second level, on the other hand, is tied to the details of the architecture being specified.

Let us now introduce the CTL scheme formula $I(f_1, f_2)$, where f_1 and f_2 are CTL formulae expressing specific state properties. It expresses the ordering among significant states and their behavioral properties. $I(f_1, f_2)$ is read as follows.

> For every state s_i, if s_i satisfies f_1, then every path starting from s_i holds f_1 until a state s_k that does not satisfy f_1 is reached, such that every path starting from s_k holds $\neg f_1$ until a state that satisfies f_2 is reached.

We use the formula to model two different situations:

1. a component is in a state in which it can input data from another component (this is encoded in f_1) and then eventually consumes the data (this is encoded in f_2); and
2. a component is in a state in which it can input data from another component (this is encoded in f_1), and then it eventually consumes the data and produces an output (this is encoded in f_2).

In other words, the difference between the two situations above is that in one case we use f_2 to model the state of a component in which it has taken the input but not yet produced the output, whereas in the other case we use f_2 to model the fact that the component has actually produced

the output after having consumed the input. These two situations are different, since they occur at different states in the component's lifetime and, therefore, at different points in the computation.

This is sufficient to express the conditions/constraints imposed by the functional specification in terms of the actual computations of the architectures. Again, the final formula that expresses the correctness of the whole architecture will be a suitable combination, by means of conjunctions or disjunctions, of scheme formulae.

Causal dependencies are expressed as a conjunction of scheme formulae. They are of the form $I(f_1, f_2) \wedge I(f_2, f_3)$. Note that the two formulae share the atomic proposition f_2. Depending on whether we are modeling case 1 or case 2 above, we can use the scheme formula in two different ways: Either it expresses a dependency relative to the same component, in which the conjunction implies that the component receives data in two different steps (one encoded in f_1 the other in f_2), or it expresses a dependency between two different components, in which the conjunction implies that the component associated with $I(f_1, f_2)$ passes data to the component associated with $I(f_2, f_3)$.

Concurrency is expressed as a disjunction of scheme formulae. This is because parallelism in the architectural specifications is modeled via interleaving. Thus, the possible parallel execution between two events e_1 and e_2 will be modeled by saying that e_1 is followed by e_2, or e_2 is followed by e_1.

We thus have a formal framework in which to relate the functional and architectural specifications. In the next section we introduce the architectural variants that are the object of our study and then show how to validate those specifications with respect to the desired coherency property.

3 Architectural Specifications of AGST

The AEGIS GeoServer Simulation Testbed lends itself to a variety of applications of software architecture technology, including specification, analysis, and simulation. For example, Allen and Garlan give two architectural specifications for the system using their language Wright [2]. They show how an analysis of the first specification can reveal the potential deadlock previously discovered by the designers of the system. They then specify the designers' revision of the architecture that avoids the deadlock.

Here we are concerned with the global invariant described in the previous section. We provide a specification of the basic architecture of AGST informally described by Balzer, followed by specifications of three variants of that architecture intended to satisfy the coherency constraint in three quite different ways. The challenge is to prove that they indeed satisfy the invariant, showing how they accomplish this under their particular design choices.

3.1 Base Architecture

The three variants of the AGST architecture can best be understood in terms of how they differ from a common base architecture specification. This specification can be given at a level of abstraction that captures the essential elements and interconnections described by Balzer. In a sense, it represents the obvious interpretation of his informal description, but without considering the constraints necessary for correct operation.

Figure 2 depicts the base architecture. The four functional elements described in the previous section are realized as separate architectural components. To these are added three other components: Doctrine Authoring (DA), Track Server (TS), and Experiment Control (EC).[2] DA maintains

[2]Again, the names that we were given are not particularly consistent with the roles that those elements play within

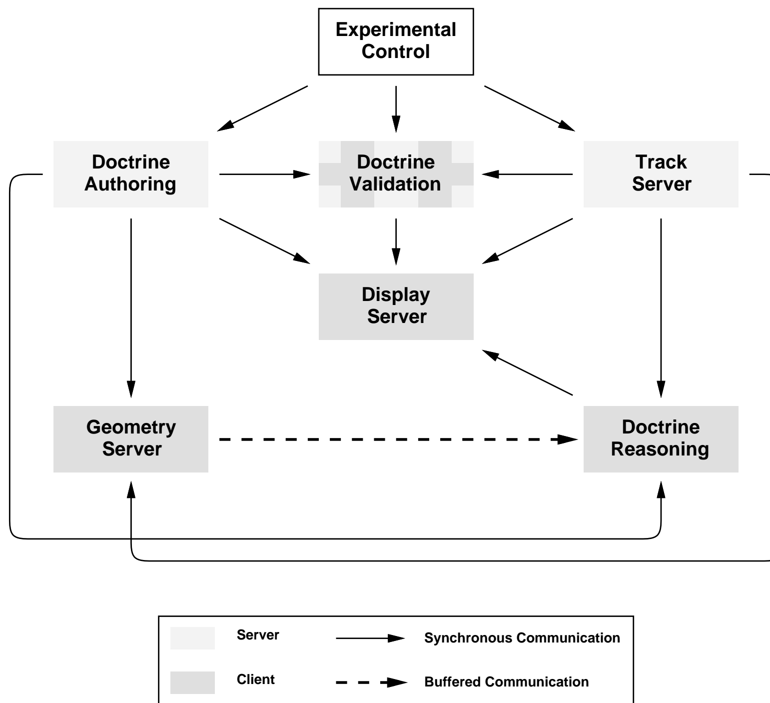

Figure 2: Basic Architecture of AGST.

a database of doctrines, while TS maintains a database of tracks. EC supplies initialization data in a single communication to DA, DV, and TS, giving *rules* of engagement to DA (from which it computes doctrines), region responsibility *commands* to DV, and the tracks of moving objects to TS. Once initialized, DA, DV, and TS act as servers for the clients of their information. DV, because it requires information from DA and TS, acts as both a server and a client. GS operates asynchronously with respect to DR and so uses a buffer to store intersections provided to DR. DR, in turn, "pushes" actions toward DS, which causes a ripple effect of requests from DA, DV, and TS. The simulation is in some sense driven by GS, with its asynchronous production of intersections based on the available doctrine and track information.

Although there are only seven architectural components in this system, there are several different kinds of communication and synchronization relationships among the elements, which makes this an interesting example.

We develop the base architecture specification in CHAM (see appendix section A.1) using a three-step process. First, we define an algebra of molecules that gives the syntax by which molecules can be built. Second, we define an initial solution, which is a subset of all possible molecules that can be constructed using the syntax and which corresponds to the initial, static configuration of a system conforming to the architecture. Finally, we define the transformation rules that represent how the system can dynamically evolve.

the system. The shading used in Figure 2 is a more accurate indication of roles.

3.1.1 Syntax

The syntax for the base architecture consists of a set of constants P representing the processing elements and a set of constants D representing the data elements. The connecting elements are given by a third set C consisting of communication, synchronization, and concurrency operations. The syntax Σ_{base} of molecules M is given by the following.

$$
\begin{aligned}
M &::= P \mid C \mid M \diamond M \mid M^\star \mid M \parallel M \\
P &::= \mathsf{EC} \mid \mathsf{DV} \mid \mathsf{DS} \mid \mathsf{DA} \mid \mathsf{TS} \mid \mathsf{GS} \mid \mathsf{DR} \\
D &::= \textit{intersection} \mid \textit{track} \mid \textit{action} \mid \textit{map} \mid \textit{rule} \mid \\
&\qquad \textit{command} \mid \textit{doctrine} \mid \mathsf{signal} \mid \{D'\} \\
D' &::= D \mid D, D' \\
C &::= \mathsf{input}(P,D,P) \mid \mathsf{output}(P,D,P) \mid \mathsf{connect}(P,P) \mid \mathsf{disconnect}(P,P) \mid \\
&\qquad \mathsf{open}(P) \mid \mathsf{join} \mid \mathsf{closed} \mid \mathsf{request}(P,D,P) \mid \mathsf{serve}(P) \mid \\
&\qquad \mathsf{buffer}(P,D,P) \mid P.P.\mathsf{Buffer} \mid C + C
\end{aligned}
$$

The construct $\{D'\}$ allows the formation of sets of data elements. We use this below to create both homogeneous and heterogeneous sets.

3.1.2 Initial Solution

The initial solution for the base architecture is as follows.

$$
\begin{aligned}
S_1 = \; & (\mathsf{output}(\mathsf{DA},\{\textit{rule},\ldots\},\mathsf{EC}) + \mathsf{output}(\mathsf{DV},\{\textit{command},\ldots\},\mathsf{EC}) \\
& \quad + \mathsf{output}(\mathsf{TS},\{\textit{track},\ldots\},\mathsf{EC})) \diamond \mathsf{EC}, \\
& \mathsf{input}(\mathsf{EC},\{\textit{rule},\ldots\},\mathsf{DA}) \diamond \mathsf{open}(\mathsf{DA}) \diamond \mathsf{closed} \diamond \mathsf{DA}, \\
& \mathsf{input}(\mathsf{EC},\{\textit{track},\ldots\},\mathsf{TS}) \diamond \mathsf{open}(\mathsf{TS}) \diamond \mathsf{closed} \diamond \mathsf{TS}, \\
& \mathsf{input}(\mathsf{EC},\{\textit{command},\ldots\},\mathsf{DV}) \diamond (\mathsf{connect}(\mathsf{DA},\mathsf{DV}) + \mathsf{connect}(\mathsf{TS},\mathsf{DV})) \\
& \quad \diamond \mathsf{open}(\mathsf{DV}) \diamond \mathsf{closed} \diamond (\mathsf{disconnect}(\mathsf{DA},\mathsf{DV}) + \mathsf{disconnect}(\mathsf{TS},\mathsf{DV})) \diamond \mathsf{DV}, \\
& (\mathsf{connect}(\mathsf{DA},\mathsf{GS}) + \mathsf{connect}(\mathsf{TS},\mathsf{GS})) \\
& \quad \diamond ((\mathsf{request}(\mathsf{DA},\textit{doctrine},\mathsf{GS}) + \mathsf{request}(\mathsf{TS},\textit{track},\mathsf{GS})) \diamond \mathsf{buffer}(\mathsf{DR},\textit{intersection},\mathsf{GS}))^\star \\
& \quad \diamond (\mathsf{disconnect}(\mathsf{DA},\mathsf{GS}) + \mathsf{disconnect}(\mathsf{TS},\mathsf{GS})) \diamond \mathsf{GS}, \\
& (\mathsf{connect}(\mathsf{DA},\mathsf{DR}) + \mathsf{connect}(\mathsf{TS},\mathsf{DR})) \\
& \quad \diamond (\mathsf{input}(\mathsf{GS},\textit{intersection},\mathsf{DR}) \diamond (\mathsf{request}(\mathsf{DA},\textit{doctrine},\mathsf{DR}) + \mathsf{request}(\mathsf{TS},\textit{track},\mathsf{DR})) \\
& \qquad \diamond \mathsf{output}(\mathsf{DS},\textit{action},\mathsf{DR}))^\star \\
& \quad \diamond (\mathsf{disconnect}(\mathsf{DA},\mathsf{DR}) + \mathsf{disconnect}(\mathsf{TS},\mathsf{DR})) \diamond \mathsf{DR}, \\
& (\mathsf{connect}(\mathsf{DA},\mathsf{DS}) + \mathsf{connect}(\mathsf{TS},\mathsf{DS}) + \mathsf{connect}(\mathsf{DV},\mathsf{DS})) \\
& \quad \diamond (\mathsf{input}(\mathsf{DR},\textit{action},\mathsf{DS}) \\
& \quad \diamond (\mathsf{request}(\mathsf{DV},\textit{map},\mathsf{DS}) + \mathsf{request}(\mathsf{DA},\textit{doctrine},\mathsf{DS}) + \mathsf{request}(\mathsf{TS},\textit{track},\mathsf{DS})))^\star \\
& \quad \diamond (\mathsf{disconnect}(\mathsf{DA},\mathsf{DS}) + \mathsf{disconnect}(\mathsf{TS},\mathsf{DS}) + \mathsf{disconnect}(\mathsf{DV},\mathsf{DS})) \diamond \mathsf{DS}, \\
& \mathsf{GS.DR.Buffer}
\end{aligned}
$$

This solution consists of eight molecules, one for the initial states of each of the seven major processing elements in the architecture plus one for the initial state of the buffer.

The notation $\{D,\ldots\}$ is used to indicate a finite, arbitrary length, homogeneous set of instances of D. The infix operator "$\diamond$" is used to express the state of a molecule with respect to its commu-

nication behavior. The state is understood by reading the molecule from left to right, with the left operand of the left-most "$\diamond$" operator representing the next action that the molecule is prepared to take. For convenience, we also use the operator "$\diamond$" to mark a molecule by the processing element it represents. Thus, for example, the molecules appearing in the initial solution all have the corresponding element of P as the right operand of their right-most "$\diamond$" operator.

Before describing the operators and operations in detail, let us look at an example, the molecule representing the initial state of DV. Note that the interpretation of all symbols, including the operators and operations, comes only from the transformation rules. Here we are informally anticipating their meaning.

$$\begin{gathered}\mathsf{input}(\mathsf{EC},\{\mathit{command},\ldots\},\mathsf{DV}) \diamond (\mathsf{connect}(\mathsf{DA},\mathsf{DV}) + \mathsf{connect}(\mathsf{TS},\mathsf{DV})) \\ \diamond\ \mathsf{open}(\mathsf{DV}) \diamond \mathsf{closed} \diamond (\mathsf{disconnect}(\mathsf{DA},\mathsf{DV}) + \mathsf{disconnect}(\mathsf{TS},\mathsf{DV})) \diamond \mathsf{DV}\end{gathered}$$

This molecule represents the fact that DV must first receive initialization data from EC, then establish connections to servers DA and TS, and then make itself available for connection as a server. Eventually, DV must close its service and disconnect from DA and TS.

Notice that the operator "$\diamond$" is being used to represent sequential behavior within a molecule. This contrasts with the situation among molecules within the larger solution, which is inherently a concurrent behavior, as discussed further below.

3.1.3 Transformation Rules

The semantics for the operators and operations are given by the transformation rules, where $m, m_1, \ldots \in M$, $p, p_1, \ldots \in P$, $c, c_1, \ldots \in C$, and $d \in D$.

$$
\begin{array}{lll}
T_1 & \equiv & m_1 \parallel m_2 \longrightarrow m_1,\ m_2 \\
T_2 & \equiv & (c_1 + c_2) \diamond m \longrightarrow c_1 \diamond c_2 \diamond m \\
T_3 & \equiv & (c_1 + c_2) \diamond m \longrightarrow c_2 \diamond c_1 \diamond m \\
T_4 & \equiv & (m_1)^\star \diamond m_2 \diamond \mathsf{GS} \longrightarrow m_1 \diamond (m_1)^\star \diamond m_2 \diamond \mathsf{GS} \\
T_5 & \equiv & (m_1)^\star \diamond m_2 \diamond \mathsf{GS} \longrightarrow m_2 \diamond \mathsf{GS} \\
T_6 & \equiv & (m_1)^\star \diamond m_2 \diamond \mathsf{DR},\ m_3 \diamond \mathsf{GS.DR.Buffer} \longrightarrow \\
 & & \quad m_1 \diamond (m_1)^\star \diamond m_2 \diamond \mathsf{DR},\ m_3 \diamond \mathsf{GS.DR.Buffer} \\
T_7 & \equiv & (m_1)^\star \diamond m_2 \diamond \mathsf{DS},\ \mathsf{output}(\mathsf{DS},d,\mathsf{DR}) \diamond m_3 \longrightarrow \\
 & & \quad m_1 \diamond (m_1)^\star \diamond m_2 \diamond \mathsf{DS},\ \mathsf{output}(\mathsf{DS},d,\mathsf{DR}) \diamond m_3 \\
T_8 & \equiv & \mathsf{input}(p_2,d,p_1) \diamond m_1,\ \mathsf{output}(p_1,d,p_2) \diamond m_2 \longrightarrow m_1,\ m_2 \\
T_9 & \equiv & \mathsf{buffer}(p_2,d,p_1) \diamond m_1,\ p_1.p_2.\mathsf{Buffer} \longrightarrow \\
 & & \quad m_1,\ \mathsf{output}(p_2,d,p_1) \diamond p_1.p_2.\mathsf{Buffer} \\
T_{10} & \equiv & \mathsf{buffer}(p_2,d,p_1) \diamond m_1,\ m_2 \diamond p_1.p_2.\mathsf{Buffer} \longrightarrow \\
 & & \quad m_1,\ m_2 \diamond \mathsf{output}(p_2,d,p_1) \diamond p_1.p_2.\mathsf{Buffer} \\
T_{11} & \equiv & \mathsf{connect}(p_2,p_1) \diamond m_1,\ \mathsf{open}(p_2) \diamond m_2 \longrightarrow \\
 & & \quad m_1,\ \mathsf{open}(p_2) \diamond \mathsf{join} \diamond m_2 \parallel (\mathsf{serve}(p_1))^\star \diamond p_2 \\
T_{12} & \equiv & \mathsf{disconnect}(p_2,p_1) \diamond m_1,\ (\mathsf{serve}(p_1))^\star \diamond p_2,\ \mathsf{open}(p_2) \diamond \mathsf{join} \diamond m_2 \longrightarrow \\
 & & \quad m_1,\ \mathsf{open}(p_2) \diamond m_2 \\
T_{13} & \equiv & \mathsf{request}(p_2,d,p_1) \diamond m,\ (\mathsf{serve}(p_1))^\star \diamond p_2 \longrightarrow \\
 & & \quad \mathsf{output}(p_2,\mathsf{signal},p_1) \diamond \mathsf{input}(p_2,d,p_1) \diamond m,\ \mathsf{serve}(p_1) \diamond (\mathsf{serve}(p_1))^\star \diamond p_2 \\
T_{14} & \equiv & \mathsf{serve}(p) \diamond m \diamond \mathsf{DA} \longrightarrow \mathsf{input}(p,\mathsf{signal},\mathsf{DA}) \diamond \mathsf{output}(p,\mathit{doctrine},\mathsf{DA}) \diamond m \diamond \mathsf{DA} \\
T_{15} & \equiv & \mathsf{serve}(p) \diamond m \diamond \mathsf{TS} \longrightarrow \mathsf{input}(p,\mathsf{signal},\mathsf{TS}) \diamond \mathsf{output}(p,\mathit{track},\mathsf{TS}) \diamond m \diamond \mathsf{TS} \\
T_{16} & \equiv & \mathsf{serve}(p) \diamond m \diamond \mathsf{DV} \longrightarrow \\
 & & \quad \mathsf{input}(p,\mathsf{signal},\mathsf{DV}) \\
 & & \quad \diamond\ (\mathsf{request}(\mathsf{DA},\mathit{doctrine},\mathsf{DV}) + \mathsf{request}(\mathsf{TS},\mathit{track},\mathsf{DV})) \\
 & & \quad \diamond\ \mathsf{output}(p,\mathit{map},\mathsf{DV}) \diamond m \diamond \mathsf{DV} \\
T_{17} & \equiv & \mathsf{GS},\ \mathsf{GS.DR.Buffer},\ (m_1)^\star \diamond m_2 \diamond \mathsf{DR},\ (m_3)^\star \diamond m_4 \diamond \mathsf{DS} \longrightarrow \\
 & & \quad \mathsf{GS},\ \mathsf{GS.DR.Buffer},\ m_2 \diamond \mathsf{DR},\ m_4 \diamond \mathsf{DS} \\
T_{18} & \equiv & \mathsf{DS},\ \mathsf{open}(\mathsf{DV}) \diamond \mathsf{closed} \diamond m \diamond \mathsf{DV} \longrightarrow \mathsf{DS},\ m \diamond \mathsf{DV} \\
T_{19} & \equiv & \mathsf{DV},\ \mathsf{open}(\mathsf{DA}) \diamond \mathsf{closed} \diamond \mathsf{DA},\ \mathsf{open}(\mathsf{TS}) \diamond \mathsf{closed} \diamond \mathsf{TS} \longrightarrow \mathsf{DV},\ \mathsf{DA},\ \mathsf{TS}
\end{array}
$$

The rules are of two kinds. The first kind defines the basic language of the abstract machine. They effectively augment the primitive rules defined for all CHAMs, specifically targeted for the application at hand. Their role is simply to define the syntactic properties of the given operators—such as their translation into expressions involving other, presumably lower-level operators—without introducing any behavior (i.e., state change) into the system. Rules of the second kind, in contrast, define the dynamics of the system itself. Their application indicates actual changes in the state of the modeled system. Thus, rules of the second kind represent true computational progress, while rules of the first kind represent allowable and convenient restructurings of CHAM expressions.

Rules T_1 through T_3 are of the first kind. The infix operator "$\parallel$" syntactically represents a complexly composed molecule that can be broken down into parallel subcomponents, thus allowing multiple reactions to occur simultaneously. In more familiar terms, "$\parallel$" can be intuitively interpreted as a parallel operator. T_1 accomplishes this by placing the constituent molecules into the larger, inherently concurrent, solution. Rules T_2 and T_3 define the infix operator "$+$" in terms of "$\diamond$". In particular, we can see that "$+$" is a nondeterministic ordering of c_1 and c_2; either can occur before the other, but both must eventually occur.

Iterative behavior within molecules is expressed through the unary operator "$\star$". For example, the molecule representing the initial state of GS

$$
\begin{aligned}
&(\mathsf{connect}(\mathsf{DA},\mathsf{GS}) + \mathsf{connect}(\mathsf{TS},\mathsf{GS})) \\
\diamond\ &((\mathsf{request}(\mathsf{DA},\mathit{doctrine},\mathsf{GS}) + \mathsf{request}(\mathsf{TS},\mathit{track},\mathsf{GS})) \diamond \mathsf{buffer}(\mathsf{DR},\mathit{intersection},\mathsf{GS}))^\star \\
\diamond\ &(\mathsf{disconnect}(\mathsf{DA},\mathsf{GS}) + \mathsf{disconnect}(\mathsf{TS},\mathsf{GS})) \diamond \mathsf{GS}
\end{aligned}
$$

represents the fact that GS must first establish a connection to servers DA and TS, then iteratively request data from those servers and place an intersection into the buffer for DR. Once the iterative behavior ends, GS must disconnect from the servers. While an iterant itself represents behavior (e.g., the GS behavior of requesting data from DA and TS, and then placing data into a buffer), the syntactic creation of that iterant within the solution is not considered behavior. Thus, rules T_4 through T_7 are of the first kind, specifying iterations of behavior within the GS, DR, and DS molecules by making available an iterant, m_1, from within the context of an iteration expression, $(m_1)^\star$.

As mentioned in Section 2, the system is driven by the behavior of GS. T_4 allows arbitrarily many iterations of the behavior m_1. T_5, which has the same precondition as T_4, is non-deterministically chosen to indicate that GS terminates its iterative behavior. T_6 provides the context for iteration within DR, namely if there are data waiting in the buffer between GS and DR. T_7 provides the corresponding context for DS.

The next set of rules are of the second kind, defining the dynamics of the system. We begin with T_8, the basic communication rule. This rule indicates the synchronization and transfer of data between molecules representing two different processing elements. The operations input and output represent communication ports, where the targets of the communication and the data to be exchanged are explicitly named. The rule indicates that if two molecules of the form given to the left of the arrow exist in the solution, then they can be rewritten as the two molecules of the form given to the right of the arrow.

Buffered communication is represented by the operation buffer, which places a data element into the buffer placed between a particular pair of processing elements. The buffer between two processing elements is represented by the construct $P.P.$Buffer; the first processing element in the construct is understood to be the producer and the second to be the consumer. The initial solution, S_1, contains such a buffer for GS, the producer, and DR, the consumer. T_9 and T_{10} define the semantics of buffering in terms of the addition of a basic output opportunity to an ordered list of such opportunities maintained by the buffer molecule. T_9 addresses the case of an empty buffer, while T_{10} addresses the case for a non-empty buffer.

We turn now to the representation of the client-server behavior. This kind of communication involves a two-level protocol, whereby a connection is first established, and then a series of synchronized request and serve operations are performed to effect the actual data transfer. The connection is then dismantled. This common protocol is illustrated in Figure 3, which uses a sequence chart to show two clients interacting with a shared server over time, where time progresses from top to bottom in the figure.

The server contains multiple threads of control, one for the server to accept connections, plus one for each client to serve that client's requests.[3] The creation of a new thread is represented by T_{11}. The operator "$\|$" is used to indicate that the service thread operates in parallel with the other molecules. The service thread is also given the obligation, through the presence of the operator close, to eventually close down after an arbitrary number of requests have been served. The

[3]This is an answer to the deadlock problem examined by Allen and Garlan [2].

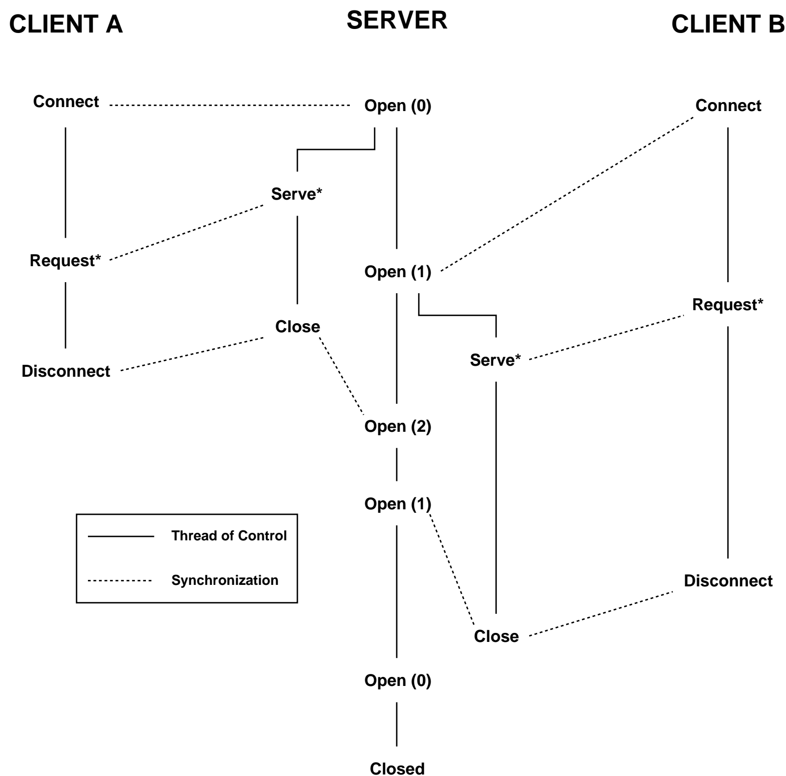

Figure 3: Client-Server Protocol.

operator join, which is added to the main server molecule, acts as a counter of the open connections. This is shown in Figure 3 as the number to the right of open. The idea is that the thread that is "forked" for a client must eventually "join" when the connection is closed down, and that all such connections must be closed before the server itself can shut down. This is represented by T_{12}, which indicates a synchronization among three molecules to effect the dismantling of a connection. Notice that the service molecule, and hence the thread it represents, is removed from the solution by the application of T_{12}.

The rule for requesting data from a server, T_{13}, and three rules for serving data to a client, T_{14} through T_{16}, are all rules of the first kind mentioned above. In particular, operations request and serve are notational shorthands introduced solely to improve the readability of the specification. The rules give the definition of these shorthands in terms of the behavior-exhibiting operations input and output. In essence, they show that request and serve are a handshake protocol involving paired input and output, first of a signal indicating the request, and then of a data transfer that is the actual serving of data.

Why are there three server-specific rules for serving data and not just one, generic rule as there is for a request? The reason is that, unlike DA and TS, DV is not a simple server. It must engage in some additional communication in order to service each request. This difference can be clearly seen in rule T_{16}, which indicates the mixed server/client behavior of DV (c.f., Figure 2).

To anticipate the discussion of the three variants in the next section, we note that rules T_{13} through T_{16} are the critical rules that reflect the differences among the architectures. In particular,

the first variant modifies all four rules, the second variant modifies rules T_{13} and T_{16}, and the third variant modifies rules T_{14} and T_{15}. This should not be surprising, since the essence of the variation has to do with the treatment of servers DA and TS, but it does highlight our ability to isolate critical aspects of an architecture using this specification technique.

The next three rules, T_{17} through T_{19}, represent the orderly shutdown of the system. The shutdown is initiated by the termination of GS, which in that state is considered “inert” because no rules can be applied to the molecule to further change its state. In particular, it can no longer produce data for the GS/DR buffer, and eventually this leads to the emptying of that buffer. T_{17} gives the context by which DR and DS can begin to shutdown, namely when GS terminates and the buffer is empty. T_{18} gives the corresponding context for DV, namely that DS has terminated. Finally, T_{19} indicates the condition under which the entire system completes execution, namely that all seven processing elements individually become inert and the buffer is empty.

3.2 Three Architectural Variants

Any specification of a system will reflect the deliberate choice of a particular level of detail. The specification of the base architecture given above captures the basic modularization and interactions among the elements, but purposely leaves unstated many other aspects of the system. One aspect of concern to the designers, as mentioned in Section 2, is whether and how GS, DR, DS, and DV coordinate their computations involving doctrines and tracks. The designers proposed three alternative architectures, which we have modeled in CHAM and describe in this section in terms of their differences from the base architecture. The full specifications appear in Appendix B.

3.2.1 Identifiers

The first variant attempts to achieve the coherent processing of data by associating unique identifiers with each doctrine and track. GS assigns these identifiers when it first requests them for its computation of an intersection. The identifiers are then packaged with the computed intersection and placed in the GS/DR buffer. DR, DS, and DV use the identifiers when retrieving doctrines and tracks.

The only change needed to the base syntax, Σ_{base}, for use in the specification of this architecture is to replace the data element signal of D with the data element *identifier*.

The change to the initial solution, S_1, involves changes to the initial molecules of GS, DR, and DV. In essence what is required is that all requests for doctrines and tracks from DA and TS, respectively, must be enhanced to include the use of identifiers. Moreover, the output from GS, DR, and DV must provide the associated identifiers for the doctrines and tracks used to calculate their functional output.

First, consider the initial molecule for GS.

$$\begin{aligned}
&(\mathsf{connect(DA,GS)} + \mathsf{connect(TS,GS)}) \\
\diamond\ &((\mathsf{request(DA,}\{\mathit{identifier}_1,\mathit{doctrine}\}\mathsf{,GS)} + \mathsf{request(TS,}\{\mathit{identifier}_2,\mathit{track}\}\mathsf{,GS)}) \\
&\quad \diamond\ \mathsf{buffer(DR,}\{\mathit{identifier}_1,\mathit{identifier}_2,\mathit{intersection}\}\mathsf{,GS)})^{\star} \\
\diamond\ &(\mathsf{disconnect(DA,GS)} + \mathsf{disconnect(TS,GS)}) \diamond \mathsf{GS}
\end{aligned}$$

The requests for doctrines and tracks, in addition to requesting the data elements themselves, assign the identifiers by which they can be subsequently referenced. (We are assuming here that DA and TS will always provide the same data upon being presented with a request for those same identifiers. This assumption about the correct internal behavior of components is reasonable at

the architectural level of concern. Of course, the validity of that assumption must be established, but not as part of architecture-level analysis.) The output from GS, which is placed in a buffer, is a triple consisting of an intersection and the two identifiers of the doctrine and track used to calculate that intersection.

Next, consider the initial molecule for DR.

$$
\begin{aligned}
&(\mathsf{connect}(\mathsf{DA},\mathsf{DR}) + \mathsf{connect}(\mathsf{TS},\mathsf{DR}))\\
\diamond\ &(\mathsf{input}(\mathsf{GS},\{\mathit{identifier}_1,\mathit{identifier}_2,\mathit{intersection}\},\mathsf{DR})\\
&\quad \diamond (\mathsf{request}(\mathsf{DA},\{\mathit{identifier}_1,\mathit{doctrine}\},\mathsf{DR}) + \mathsf{request}(\mathsf{TS},\{\mathit{identifier}_2,\mathit{track}\},\mathsf{DR}))\\
&\quad \diamond \mathsf{output}(\mathsf{DS},\{\mathit{identifier}_1,\mathit{identifier}_2,\mathit{action}\},\mathsf{DR}))^\star\\
\diamond\ &(\mathsf{disconnect}(\mathsf{DA},\mathsf{DR}) + \mathsf{disconnect}(\mathsf{TS},\mathsf{DR})) \diamond \mathsf{DR}
\end{aligned}
$$

DR obtains an intersection and the identifiers for the doctrine and track used to calculate that intersection. Those identifiers are subsequently used to request the corresponding data elements from DA and TS. Like GS, the output of DR is a triple that includes the identifiers.

Finally, consider the initial molecule for DS.

$$
\begin{aligned}
&(\mathsf{connect}(\mathsf{DA},\mathsf{DS}) + \mathsf{connect}(\mathsf{TS},\mathsf{DS}) + \mathsf{connect}(\mathsf{DV},\mathsf{DS}))\\
\diamond\ &(\mathsf{input}(\mathsf{DR},\{\mathit{identifier}_1,\mathit{identifier}_2,\mathit{action}\},\mathsf{DS})\\
&\quad \diamond (\mathsf{request}(\mathsf{DV},\{\{\mathit{identifier}_1,\mathit{identifier}_2\},\mathit{map}\},\mathsf{DS}) + \mathsf{request}(\mathsf{DA},\{\mathit{identifier}_1,\mathit{doctrine}\},\mathsf{DS})\\
&\qquad + \mathsf{request}(\mathsf{TS},\{\mathit{identifier}_2,\mathit{track}\},\mathsf{DS})))^\star\\
\diamond\ &(\mathsf{disconnect}(\mathsf{DA},\mathsf{DS}) + \mathsf{disconnect}(\mathsf{TS},\mathsf{DS}) + \mathsf{disconnect}(\mathsf{DV},\mathsf{DS})) \diamond \mathsf{DS}
\end{aligned}
$$

The changes are quite similar to those made for DR, the only substantial difference being the need to provide the two identifiers to DV as part of the request for a map.

To accommodate the use of identifiers, rules T_{13} through T_{16} are the only transformation rules of the base architecture specification that require modification. These four rules are the ones that model the general request protocol for clients and the specific serve protocol for the servers DA, TS, and DV.

$$
\begin{aligned}
T_{13} \equiv\ & \mathsf{request}(p_2,\{d_1,d_2\},p_1) \diamond m,\ (\mathsf{serve}(p_1))^\star \diamond p_2 \longrightarrow\\
& \quad \mathsf{output}(p_2,d_1,p_1) \diamond \mathsf{input}(p_2,d_2,p_1) \diamond m,\ \mathsf{serve}(p_1) \diamond (\mathsf{serve}(p_1))^\star \diamond p_2\\
T_{14} \equiv\ & \mathsf{serve}(p) \diamond m \diamond \mathsf{DA} \longrightarrow \mathsf{input}(p,\mathit{identifier},\mathsf{DA}) \diamond \mathsf{output}(p,\mathit{doctrine},\mathsf{DA}) \diamond m \diamond \mathsf{DA}\\
T_{15} \equiv\ & \mathsf{serve}(p) \diamond m \diamond \mathsf{TS} \longrightarrow \mathsf{input}(p,\mathit{identifier},\mathsf{TS}) \diamond \mathsf{output}(p,\mathit{track},\mathsf{TS}) \diamond m \diamond \mathsf{TS}\\
T_{16} \equiv\ & \mathsf{serve}(p) \diamond m \diamond \mathsf{DV} \longrightarrow\\
& \quad \mathsf{input}(p,\{\mathit{identifier}_1,\mathit{identifier}_2\},\mathsf{DV})\\
& \quad \diamond (\mathsf{request}(\mathsf{DA},\{\mathit{identifier}_1,\mathit{doctrine}\},\mathsf{DV}) + \mathsf{request}(\mathsf{TS},\{\mathit{identifier}_2,\mathit{track}\},\mathsf{DV}))\\
& \quad \diamond \mathsf{output}(p,\mathit{map},\mathsf{DV}) \diamond m \diamond \mathsf{DV}
\end{aligned}
$$

In this variant, requests involve pairs of data elements, one of which is the identifier and the other of which is the actual data. The servers will first input an identifier and then output the corresponding data. In the case of DV, the data are the maps that it computes.

3.2.2 Bundles

The second variant attempts to achieve the coherent processing of data by having the doctrines and tracks bundled together with the results of computations that use them. So, for example, rather than packaging identifiers with intersections, as in the first variant, GS packages doctrines and tracks with intersections and then places the packages into the GS/DR buffer. DR in turn

packages the doctrines and tracks it retrieves from the buffer together with the action it computes, and passes this bundle onto DS.

No changes to the base syntax, Σ_{base}, are required to model this variant.

The changes to the initial solution, S_1, are of two kinds. The first is to remove the connections from DV, DS, and DR to the doctrine and track servers DA and TS. These connections are not required, since the doctrines and tracks will be provided to these processing elements through other means. Only GS will still make requests of DA and TS. The second change is to pass doctrines and tracks directly between the components, in essence having them flow through the system along with the data to whose computation they contributed. The initial molecules for GS, DR, and DS are as follows.

$$
\begin{aligned}
&(\mathsf{connect}(\mathsf{DA},\mathsf{GS}) + \mathsf{connect}(\mathsf{TS},\mathsf{GS}))\\
\diamond\ &((\mathsf{request}(\mathsf{DA},\mathit{doctrine},\mathsf{GS}) + \mathsf{request}(\mathsf{TS},\mathit{track},\mathsf{GS}))\\
\diamond\ &\mathsf{buffer}(\mathsf{DR},\{\mathit{doctrine},\mathit{track},\mathit{intersection}\},\mathsf{GS}))^\star\\
\diamond\ &(\mathsf{disconnect}(\mathsf{DA},\mathsf{GS}) + \mathsf{disconnect}(\mathsf{TS},\mathsf{GS})) \diamond \mathsf{GS}
\end{aligned}
$$

$$
\begin{aligned}
&(\mathsf{input}(\mathsf{GS},\{\mathit{doctrine},\mathit{track},\mathit{intersection}\},\mathsf{DR})\\
&\quad \diamond \mathsf{output}(\mathsf{DS},\{\mathit{doctrine},\mathit{track},\mathit{action}\},\mathsf{DR}))^\star \diamond \mathsf{DR}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{connect}(\mathsf{DV},\mathsf{DS})\\
\diamond\ &(\mathsf{input}(\mathsf{DR},\{\mathit{doctrine},\mathit{track},\mathit{action}\},\mathsf{DS}) \diamond \mathsf{request}(\mathsf{DV},\{\{\mathit{doctrine},\mathit{track}\},\mathit{map}\},\mathsf{DS}))^\star\\
\diamond\ &\mathsf{disconnect}(\mathsf{DV},\mathsf{DS}) \diamond \mathsf{DS}
\end{aligned}
$$

To adjust the transformation rules, we must add a new rule T'_{13} and modify rule T_{16}.

$$
\begin{aligned}
T'_{13} \ \equiv\ & \mathsf{request}(p_2,\{d_1,d_2\},p_1) \diamond m,\ (\mathsf{serve}(p_1))^\star \diamond p_2 \longrightarrow\\
& \qquad \mathsf{output}(p_2,d_1,p_1) \diamond \mathsf{input}(p_2,d_2,p_1) \diamond m,\ \mathsf{serve}(p_1) \diamond (\mathsf{serve}(p_1))^\star \diamond p_2\\
T_{16} \ \equiv\ & \mathsf{serve}(p) \diamond m \diamond \mathsf{DV} \longrightarrow\\
& \qquad \mathsf{input}(p,\{\mathit{doctrine},\mathit{track}\},\mathsf{DV}) \diamond \mathsf{output}(p,\mathit{map},\mathsf{DV}) \diamond m \diamond \mathsf{DV}
\end{aligned}
$$

The new rule T'_{13} is the same as the modified T_{13} of the identifier variant. It accommodates the form of requests made of DV, which requires a doctrine and track pair to compute a map. The original rule T_{13} is used for requests made of DA and TS by GS. The change to T_{16} involves removing the now unneeded interaction between DV and the servers DA and TS.

3.2.3 Lock Step

The third variant attempts to achieve the coherent processing of data in a rather different way from the other two variants. The idea behind this variant is that the global computation should proceed in discrete steps, enforcing the invariant through its exercise of control rather than through its manipulation of data. At each step, DA and TS should serve the same data in response to every request made during that step. The individual computations performed by GS, DR, DS, and DV, which together form a given step, thereby make use of the same pair of doctrine and track. In a coordinated manner, DA and TS are instructed to serve a new pair of doctrine and track at the beginning of the next step in the global computation.

This lock-step scheme is reflected in the specification through two devices. First, we associate a *current data* thread with each of DA and TS. Each DA and TS server thread (c.f., Figure 3) is made to synchronize with its component's current data thread. Second, we use two sets of *tokens*, one for each of DA and TS, to regulate when to move to the next step. Each set contains tokens for

each of the four clients GS, DR, DS, and DV. When all the tokens from both sets are exhausted, DA and TS can begin to serve a new doctrine and a new track, respectively, using a new set of tokens created to correspond to the next series of requests.

In terms of the syntax, we add four connecting elements to the set C: token(P,D), spent(P,D), step, and current(D).

The initial solution contains two changes. First, current data threads are added to the molecules for DA and TS.

$$\mathsf{input}(\mathsf{EC},\{\textit{rule},\ldots\},\mathsf{DA}) \diamond (\mathsf{open}(\mathsf{DA}) \diamond \mathsf{closed} \diamond \mathsf{DA} \parallel \mathsf{step} \diamond \mathsf{DA})$$

$$\mathsf{input}(\mathsf{EC},\{\textit{track},\ldots\},\mathsf{TS}) \diamond (\mathsf{open}(\mathsf{TS}) \diamond \mathsf{closed} \diamond \mathsf{TS} \parallel \mathsf{step} \diamond \mathsf{TS})$$

Second, tokens are added to indicate that DA and TS are yet to be requested for the first pair of data elements.

$$\mathsf{token}(\mathsf{DA},\mathsf{GS}),\ \mathsf{token}(\mathsf{DA},\mathsf{DR}),\ \mathsf{token}(\mathsf{DA},\mathsf{DV}),\ \mathsf{token}(\mathsf{DA},\mathsf{DS})$$

$$\mathsf{token}(\mathsf{TS},\mathsf{GS}),\ \mathsf{token}(\mathsf{TS},\mathsf{DR}),\ \mathsf{token}(\mathsf{TS},\mathsf{DV}),\ \mathsf{token}(\mathsf{TS},\mathsf{DS})$$

We also modify the transformation rules in two ways. First, we add two new rules to represent control over the lock-step progress of the data served by DA and TS.

$$\begin{aligned}
T_{20} \ &\equiv\ \mathsf{step} \diamond \mathsf{DA},\ \mathsf{step} \diamond \mathsf{TS} \longrightarrow \mathsf{current}(\textit{doctrine}) \diamond \mathsf{DA},\ \mathsf{current}(\textit{track}) \diamond \mathsf{TS} \\
T_{21} \ &\equiv\ \mathsf{spent}(p,\mathsf{GS}),\ \mathsf{spent}(p,\mathsf{DR}),\ \mathsf{spent}(p,\mathsf{DV}),\ \mathsf{spent}(p,\mathsf{DS}),\ \mathsf{current}(d) \diamond p \longrightarrow \\
&\qquad \mathsf{token}(p,\mathsf{GS}),\ \mathsf{token}(p,\mathsf{DR}),\ \mathsf{token}(p,\mathsf{DV}),\ \mathsf{token}(p,\mathsf{DS}),\ \mathsf{step} \diamond p
\end{aligned}$$

T_{20} models the establishment of a new current doctrine and current track. Notice that DA and TS are made to synchronize when this is done. T_{21} models the synchronization required to deactivate the current doctrine or track. Once both the current doctrine and current track have been deactivated, then a new current doctrine and track can be established through T_{20}. The second change to the transformation rules is to modify the serve rules for DA and TS.

$$\begin{aligned}
T_{14} \ &\equiv\ \mathsf{serve}(p) \diamond m \diamond \mathsf{DA},\ \mathsf{current}(\textit{doctrine}) \diamond \mathsf{DA},\ \mathsf{token}(\mathsf{DA},p) \longrightarrow \\
&\qquad \mathsf{input}(p,\mathsf{signal},\mathsf{DA}) \diamond \mathsf{output}(p,\textit{doctrine},\mathsf{DA}) \diamond m \diamond \mathsf{DA}, \\
&\qquad \mathsf{current}(\textit{doctrine}) \diamond \mathsf{DA},\ \mathsf{spent}(\mathsf{DA},p) \\
T_{15} \ &\equiv\ \mathsf{serve}(p) \diamond m \diamond \mathsf{TS},\ \mathsf{current}(\textit{track}) \diamond \mathsf{TS},\ \mathsf{token}(\mathsf{TS},p) \longrightarrow \\
&\qquad \mathsf{input}(p,\mathsf{signal},\mathsf{TS}) \diamond \mathsf{output}(p,\textit{track},\mathsf{TS}) \diamond m \diamond \mathsf{TS}, \\
&\qquad \mathsf{current}(\textit{track}) \diamond \mathsf{TS},\ \mathsf{spent}(\mathsf{TS},p)
\end{aligned}$$

These rules cause the server thread to synchronize with the current data thread and to consume a token, where consumption is represented by the term spent(p_1,p_2).

4 Validating the Architectural Specifications

Our goal in this work is to understand how to validate a state-based operational architectural specification against a functional requirement specification. We introduce our method largely by example, describing how to formally relate the functional specification given in Section 2 to the architectural specifications given in Section 3.

In the following we consider each of the three variants of AGST in turn. As mentioned above, we are assuming at this level of abstraction that components exhibit locally correct behavior,

such as that a buffer does not corrupt buffered data. This is reasonable in the context of a compositional design method, where reusable components are expected to be designed and validated independently. But from a somewhat different perspective, we observe that this assumption also can be turned around to place obligations on the validation of local behavior. In particular, the specifications at the architectural level make explicit what is required at the component level in order to ensure functionally correct behavior of the system. In any case, validation of component-local behavior is not the focus of this paper.

To proceed, we first identify those system states that represent a precise system behavior, such as the fact that GS is ready to input a doctrine and a track from DA and TS, respectively. We then try to see if there is a causal dependency between those states induced by the transformation rules. In general, this dependency does not produce a total ordering among states.

The next step is to identify properties of the portion of the computational path that contains the dependent states. In our case study it turns out that computations can be characterized by a rather regular structural pattern that is induced by the coordination policy adopted by the given architecture. Each architecture in fact adopts different ways of both representing and modifying data, by suitably coordinating the activities of the system components. Informally speaking, if a transformation rule T_i in a given architecture requires a given component C_j to be in a certain state $C_j^{s_1}$, and this can only be obtained by applying rule T_k on component C_j when it is, for example, in its initial state $C_j^{s_0}$, then this forces the transformation by the rule T_i to occur only after a transformation by rule T_k has occurred, suitably transforming the component C_j from its initial state $C_j^{s_0}$ to the state $C_j^{s_1}$. Obviously this situation has to happen in all the computations that have a transformation step obtained through the application of T_k.

As mentioned in Section 2.2, this kind of behavior can be expressed through a CTL formula that involves state-based computations, allowing for the specification of ordering among states and their properties. In this way the CTL formula aims to characterize, in terms of computations, the functionally correct behaviors. The proof that this formula is satisfied by the actual architectural specification can then be performed either automatically or by algebraic reasoning.

By combining the scheme formulae introduced in Section 2.2, we obtain a general formula that describes how the computation in the three different architectures evolves. Of course, depending on the different design decisions of the three specifications, we will have three different general formulae. Nevertheless, all of them aim at expressing the same functional specification, in the sense that they specify how causal dependencies among data and components have to be satisfied by an architectural specification. We want to make sure that along a computation the different components use proper data to proceed in their execution. For instance, we want to make sure not only that once GS has computed an intersection it is later used by DR to compute an action, but also that in doing so they use the same pair of doctrine and track. The three architectures differ mainly in the way they recover the doctrines, tracks, and maps along a computation. This is, of course, reflected in the general formulae and is the main source of differences among the three different formulations of our case study. In the case of the identifier variant, all pieces of information have a unique identity and these identities are passed along the computation. In the case of the bundle variant, all the information is bundled together. And in the lock-step variant, it is assumed that the data servers provide the same data within the same step. The assumptions behind these three architectures are discussed in more detail below.

In order to formally define our invariant $I(f_1, f_2)$ as a CTL formulae we need further notation. Consider two state formulae f_1 and f_2. Let $AG(f_1)$ abbreviate $\neg(E(true\, U\, \neg f_1))$, let $f_1 \longrightarrow f_2$ abbreviate $\neg f_1 \vee f_2$, and let $f_1 \vee f_2$ abbreviate $\neg(\neg f_1 \wedge \neg f_2)$. Then $I(f_1, f_2)$ is defined as follows.

$$I(f_1, f_2) \equiv AG(f_1 \longrightarrow A(f_1\, U\, (\neg f_1 \wedge A(\neg f_1\, U\, f_2))))$$

We also need a slightly weaker version of the scheme formula that will be useful in our formulation. Letting f_1 and f_2 be atomic formulae, then

$$J(f_1, f_2) \equiv AG(f_1 \longrightarrow A(f_1\, U\, f_2))$$

is read as follows.

> For every state s_i, if s_i satisfies f_1, then every path starting from s_i holds f_1 until a state satisfying f_2 is reached.

Moreover, rather than saying

> For every path σ starting from the initial solution s_0, if a state s_i of the path contains a set of molecules M, then every path starting from s_i has a state that contains the set of molecules M'

when stating basic dynamic properties of our architectures, we use the following, more concise phrase.

> Each state that contains a set of molecules M is followed by a state that contains a set of molecules M'.

In their current formulation, our architectural specifications are especially suitable for proving structural properties of the architectures, such as causal dependencies among components and properties related to distribution of components. However, to verify the coherency constraints on data, we need more concrete formulations. In particular, they should explicitly indicate the instance of data that DA, TS, and DS provide to their partners when queried, and should explicitly model the actual data passing in the input/output communication protocol. The required extension is simple and expected, which is the reason why we do not start directly from these more detailed descriptions. For instance, in the bundle variant we assume that in rule T_{14} (and similarly for the other rules, such as T_{15}) the doctrine variable on the right-hand side of the rule is replaced with a doctrine instance when the rule is applied. This guarantees that when an output is enabled it contains actual instances of data. Thus, the data exchange in the input/output communication protocol of rule T_8 is the expected one: the output molecule communicates the (instance of) data to the input molecule, and the latter replaces the involved variable with the received information.

4.1 Validating Bundle

We start our presentation with the bundle architecture. Let us recall the functional specification we have to express.

$$\mathsf{DS}(\mathsf{DR}(\mathsf{GS}(doctrine_i, track_j), doctrine_i, track_j), \mathsf{DV}(doctrine_i, track_j), doctrine_i, track_j)$$

In the bundle architecture, the data coherency problem is solved by packaging the data and carrying them along through the computation. The local functioning of DR, for instance, is assumed to be correct by definition; DR will compute an action by using the same doctrine and track that was used by GS to produce the intersection because the data are packaged together. What we do not know, however, is the coordination of potentially concurrent activities: is the coherency preserved when it comes time for the DS computation? In this case, DS gets data from different sources, namely DR and DV, and the fact that the data are bundled in the two respective components does not guarantee that there is consistency between the two different sets of parameters. More precisely, DR and DV can compute their result starting from different pairs of doctrine and track. This is exactly what we must be able to control and check in terms of component interactions. In order to be able to express this property in terms of a property on system computations, we follow the methodology sketched at the beginning of the section. Starting from the functional property and considering the way the bundle architecture actually computes the various pieces of data, we identify the relevant system states represented as CHAM solutions.

We start from the innermost part of the functional formula, $\mathsf{GS}(doctrine_i, track_j)$, where GS asks for a doctrine and a track from DA and TS, respectively. This is done in two possible orderings: GS asks for a doctrine from DA and then asks for a track from TS, or vice versa.

Under the first ordering, GS asks for a doctrine from DA when GS is ready to input a doctrine from DA and when DA is ready to output the doctrine to GS. This situation is possible when a solution contains both molecules μ_1 and μ_2 defined as follows.

$$\begin{aligned}\mu_1 &= \mathsf{input(DA,}\textsf{\textit{doctrine}}\mathsf{,GS)} \diamond \mathsf{request(TS,}\textsf{\textit{track}}\mathsf{,GS)} \\ &\quad\diamond \mathsf{buffer(DR,}\{\textsf{\textit{doctrine,track,intersection}}\}\mathsf{,GS)} \diamond (m_1)^\star \diamond m_2 \diamond \mathsf{GS} \\ \mu_2 &= \mathsf{output(GS,}\textsf{\textit{d}}_i\mathsf{,DA)} \diamond m_3 \diamond \mathsf{DA}\end{aligned}$$

Once GS has obtained the doctrine, it then tries to get the track from TS. GS asks for a track from TS when GS is ready to input a track from TS and TS is ready to output the track to GS. This situation is possible when a solution contains both molecules μ_3 and μ_4 defined as follows.

$$\begin{aligned}\mu_3 &= \mathsf{input(TS,}\textsf{\textit{track}}\mathsf{,GS)} \\ &\quad\diamond \mathsf{buffer(DR,}\{\textsf{\textit{doctrine,track,intersection}}\}\mathsf{,GS)} \diamond (m_1)^\star \diamond m_2 \diamond \mathsf{GS} \\ \mu_4 &= \mathsf{output(GS,}\textsf{\textit{t}}_j\mathsf{,TS)} \diamond m_4 \diamond \mathsf{TS}\end{aligned}$$

The symmetric situation happens, nondeterministically, in the other possible ordering, where GS asks first for a track and then for a doctrine. The four molecules corresponding to those above are defined as follows.

$$\begin{aligned}\mu_5 &= \mathsf{input(TS,}\textsf{\textit{track}}\mathsf{,GS)} \diamond \mathsf{request(DA,}\textsf{\textit{doctrine}}\mathsf{,GS)} \\ &\quad\diamond \mathsf{buffer(DR,}\{\textsf{\textit{doctrine,track,intersection}}\}\mathsf{,GS)} \diamond (m_1)^\star \diamond m_2 \diamond \mathsf{GS} \\ \mu_6 &= \mathsf{output(GS,}\textsf{\textit{t}}_j\mathsf{,TS)} \diamond m_4 \diamond \mathsf{TS} \\ \mu_7 &= \mathsf{input(DA,}\textsf{\textit{doctrine}}\mathsf{,GS)} \\ &\quad\diamond \mathsf{buffer(DR,}\{\textsf{\textit{doctrine,track,intersection}}\}\mathsf{,GS)} \diamond (m_1)^\star \diamond m_2 \diamond \mathsf{GS} \\ \mu_8 &= \mathsf{output(GS,}\textsf{\textit{d}}_i\mathsf{,DA)} \diamond m_3 \diamond \mathsf{DA}\end{aligned}$$

Property 1 *Each solution that contains μ_1 and μ_2 is followed by a solution that contains μ_3 and μ_4. Each solution that contains μ_5 and μ_6 is followed by a solution that contains μ_7 and μ_8.*

Once GS has obtained a doctrine and track pair, namely a solution that contains either both μ_3 and μ_4 or both μ_7 and μ_8, it computes an intersection. This is the state of the system in which GS is ready to buffer an intersection computed from the doctrine and track.

$$\mu_9 = \mathsf{buffer}(\mathsf{DR},\{d_i,t_j,i_k\},\mathsf{GS}) \diamond (m_1)^\star \diamond m_2 \diamond \mathsf{GS}$$

As we mention in Section 1, we are making a black-box assumption that the intersection to be buffered in molecule μ_9 is computed from the specific doctrine and track received, namely $i_k = \mathsf{GS}(d_i, t_j)$. In other words, we are assuming at this level of analysis that the components behave correctly according to their functional specification.

Property 2 *Each solution that contains either μ_3 and μ_4 or μ_7 and μ_8 is followed by a solution that contains μ_9.*

The state in which GS buffers a doctrine, a track, and an intersection is then followed by a state in which the buffer actually contains such a triple.

$$\mu_{10} = \mathsf{output}(\mathsf{DR},\{d_i,t_j,i_k\},\mathsf{GS}) \diamond \ldots \diamond \mathsf{GS.DR.Buffer}$$
$$\mu_{11} = (m_1)^\star \diamond m_2 \diamond \mathsf{GS}$$

Molecule μ_{10} denotes the disjunction of all possible molecules obtained by replacing "..." with sequences of outputs of the form $\mathsf{output}(\mathsf{DR},\{d_i,t_j,i_k\},\mathsf{GS})$. We note that μ_{10} is actually a CTL formula when there are a finite number of doctrines and tracks, and when the buffer is bounded. This allows us to cope with any buffer content.

Property 3 *Each solution that contains μ_9 is followed by a solution that contains μ_{10} and mu_{11}.*

Intersections buffered by GS are used by DR to compute actions. DR removes such intersections from the buffer together with their corresponding doctrines and tracks. The molecules representing a state in which DR can remove data from the buffer (i.e., the solution contains μ_{12}) and the buffer is not empty (i.e., the solution contains μ_{13}) are as follows.

$$\mu_{12} = \mathsf{input}(\mathsf{GS},\{\mathit{doctrine},\mathit{track},\mathit{intersection}\},\mathsf{DR}) \diamond \mathsf{output}(\mathsf{DS},\{\mathit{doctrine},\mathit{track},\mathit{action}\},\mathsf{DR}) \diamond (m_6)^\star \diamond \mathsf{DR}$$
$$\mu_{13} = \mathsf{output}(\mathsf{DR},\{d_i,t_j,i_k\},\mathsf{GS}) \diamond \ldots \diamond \mathsf{GS.DR.Buffer}$$

Property 4 *Each solution that contains μ_{10} and μ_{11} is followed by a solution that contains μ_{12} and μ_{13}.*

A solution that contains μ_{12} and μ_{13} is then followed by a solution in which DR provides to DS an action with the corresponding doctrine and track.

$$\mu_{14} = \mathsf{input}(\mathsf{DR},\{\mathit{doctrine},\mathit{track},\mathit{action}\},\mathsf{DS}) \diamond \mathsf{request}(\mathsf{DV},\{\{\mathit{doctrine},\mathit{track}\},\mathit{map}\},\mathsf{DS}) \diamond (m_8)^\star \diamond \mathsf{DS}$$
$$\mu_{15} = \mathsf{output}(\mathsf{DS},\{d_i,t_j,a_l\},\mathsf{DR}) \diamond (m_6)^\star \diamond \mathsf{DR}$$

Property 5 *Each solution that contains μ_{12} and μ_{13} is followed by a solution that contains μ_{14} and μ_{15}.*

This last state is followed by a graphical depiction of a map computed by DS, which we do not bother to model explicitly.

Let us now express the properties above by means of CTL. The bundle architecture must satisfy the following formula.

$$\begin{array}{l} I(\ \mu_1 \wedge \mu_2,\ \mu_3 \wedge \mu_4\ ) \wedge \\ I(\ \mu_5 \wedge \mu_6,\ \mu_7 \wedge \mu_8\ ) \wedge \\ I(\ (\mu_3 \wedge \mu_4)\ \vee\ (\mu_7 \wedge \mu_8),\ \mu_9\ ) \wedge \\ I(\ \mu_9,\ \mu_{10}\ \wedge\ \mu_{11}\ ) \wedge \\ J(\ \mu_{10}\ \wedge\ \mu_{11},\ \mu_{12}\ \wedge\ \mu_{13}\ ) \wedge \\ I(\ \mu_{12}\ \wedge\ \mu_{13},\ \mu_{14}\ \wedge\ \mu_{15}\ ) \end{array}$$

This formula explains how the computation evolves from the moment GS asks for a doctrine and a track in order to compute an intersection, to the moment DS depicts the corresponding scene. Indeed, the first two I formulae deal with the initial communications among DA, TS, and GS. The third one deals with the output of the computed intersection by GS. The fourth I formula models the placing of the intersection into the buffer. The buffered intersection constitutes the input to DR (the J formula) that is used to compute an action that is eventually sent to DS (the last I formula).

Indeed, our architecture satisfies the formula. The proof can be done either algebraically or automatically. The way we have derived properties actually provides a sketch of the proof. Only a few details on the application of the rules have been omitted. In addition, we could prove that the computation starts with a state that holds either $\mu_1 \wedge \mu_2$ or $\mu_5 \wedge \mu_6$. This can be expressed by the formula $A(\ f_1\,U\,f_2\ )$, where $f_1 = \neg(\mu_3 \wedge \mu_4) \vee (\mu_7 \wedge \mu_8)$ and $f_2 = (\mu_1 \wedge \mu_2) \vee (\mu_5 \wedge \mu_6)$. It says that in each path starting from the initial solution it cannot be the case that a state contains molecule $\mu_3 \wedge \mu_4$ $(\mu_7 \wedge \mu_8)$ before $\mu_1 \wedge \mu_2$ $(\mu_5 \wedge \mu_6)$.

4.2 Validating Lock Step

We start with an assumption appropriate to the lock-step architecture. It is related to the data that DA, TS, and DS provide to their partners when queried.

Assumption 1 *Within the same step, DA, TS, and DS each provide the same data to their clients.*

As in the bundle architecture, GS has two ways to ask for a doctrine and for a track from DA and TS, respectively: It can first ask for a doctrine from DA and then for a track from TS, or vice versa. GS asks for a doctrine from DA when GS is ready to input a doctrine from DA, when DA is ready to output the doctrine from GS, when the token from GS to DA is available, and when the current doctrine is a given d. This situation is possible when a solution contains the four molecules μ_1, μ_2, μ_3, and μ_4 defined as follows.

$$\begin{aligned} \mu_1 &= \mathsf{input}(\mathsf{DA},\mathit{doctrine},\mathsf{GS}) \diamond \mathsf{request}(\mathsf{TS},\mathit{track},\mathsf{GS}) \\ &\qquad \diamond\ \mathsf{buffer}(\mathsf{DR},\mathit{intersection},\mathsf{GS}) \diamond (m_1)^\star \diamond m_2 \diamond \mathsf{GS} \\ \mu_2 &= \mathsf{output}(\mathsf{GS},d_i,\mathsf{DA}) \diamond m_3 \diamond \mathsf{DA} \\ \mu_3 &= \mathsf{token}(\mathsf{DA},\mathsf{GS}) \\ \mu_4 &= \mathsf{current}(d_i) \diamond \mathsf{DA} \end{aligned}$$

A solution that contains those four molecules is followed by a state in which GS asks for a track. This situation is similar to the previous one. GS asks for a track from TS when GS is ready to input a track from TS, TS is ready to output the track to GS, the token from GS to TS is available, and the current track is a given t. In addition, we require that the token from GS to DA is spent

in order to make sure that in this state GS has already obtained a doctrine. This is captured with the following molecules.

$$\begin{aligned}
\mu_5 &= \mathsf{input}(\mathsf{TS},\mathit{track},\mathsf{GS}) \\
&\qquad \diamond \mathsf{buffer}(\mathsf{DR},\mathit{intersection},\mathsf{GS}) \diamond (m_1)^\star \diamond m_2 \diamond \mathsf{GS} \\
\mu_6 &= \mathsf{output}(\mathsf{GS},t_j,\mathsf{TS}) \diamond m_4 \diamond \mathsf{TS} \\
\mu_7 &= \mathsf{token}(\mathsf{TS},\mathsf{GS}) \\
\mu_8 &= \mathsf{current}(t_j) \diamond \mathsf{TS} \\
\mu_9 &= \mathsf{spent}(\mathsf{DA},\mathsf{GS})
\end{aligned}$$

Similar reasoning holds for the symmetric case when GS first asks for a track from TS and then for a doctrine from DA. This is followed by a solution in which GS asks for a doctrine from DA and the token from TS to GS is spent. The corresponding molecules are as follows.

$$\begin{aligned}
\mu_{10} &= \mathsf{input}(\mathsf{TS},\mathit{track},\mathsf{GS}) \diamond \mathsf{request}(\mathsf{DA},\mathit{doctrine},\mathsf{GS}) \\
&\qquad \diamond \mathsf{buffer}(\mathsf{DR},\mathit{intersection},\mathsf{GS}) \diamond (m_1)^\star \diamond m_2 \diamond \mathsf{GS} \\
\mu_{11} &= \mathsf{output}(\mathsf{GS},t_j,\mathsf{TS}) \diamond m_4 \diamond \mathsf{TS} \\
\mu_{12} &= \mathsf{token}(\mathsf{TS},\mathsf{GS}) \\
\mu_{13} &= \mathsf{current}(t_j) \diamond \mathsf{TS} \\
\mu_{14} &= \mathsf{input}(\mathsf{DA},\mathit{doctrine},\mathsf{GS}) \\
&\qquad \diamond \mathsf{buffer}(\mathsf{DR},\mathit{intersection},\mathsf{GS}) \diamond (m_1)^\star \diamond m_2 \diamond \mathsf{GS} \\
\mu_{15} &= \mathsf{output}(\mathsf{GS},d_i,\mathsf{DA}) \diamond m_3 \diamond \mathsf{DA} \\
\mu_{16} &= \mathsf{token}(\mathsf{DA},\mathsf{GS}) \\
\mu_{17} &= \mathsf{current}(d_i) \diamond \mathsf{DA} \\
\mu_{18} &= \mathsf{spent}(\mathsf{TS},\mathsf{GS})
\end{aligned}$$

Property 6 *Each solution that contains μ_1, μ_2, μ_3, and μ_4 is followed by a solution that contains μ_5, μ_6, μ_7, μ_8 and μ_9. Each solution that contains μ_{10}, μ_{11}, μ_{12}, and μ_{13} is followed by a solution that contains μ_{14}, μ_{15}, μ_{16}, μ_{17}, and μ_{18}.*

Once GS has obtained the current doctrine and the current track it is able to buffer an intersection. In such a state the buffer is certainly empty.

$$\begin{aligned}
\mu_{19} &= \mathsf{buffer}(\mathsf{DR},i_k,\mathsf{GS}) \diamond (m_1)^\star \diamond m_2 \diamond \mathsf{GS} \\
\mu_{20} &= \mathsf{GS.DR.Buffer}
\end{aligned}$$

Assumption 2 *Since the components locally behave correctly according to their functional specification, we can assume that in molecule μ_{19} the intersection to be buffered is computed from the doctrine and track just received, namely $i_k = GS(d_i, t_j)$.*

Property 7 *Each solution that contains either μ_5, μ_6, μ_7, μ_8, and μ_9 or μ_{14}, μ_{15}, μ_{16}, μ_{17}, and μ_{18} is followed by a solution that contains μ_{19} and μ_{20}.*

Then GS can buffer the intersection.

$$\begin{aligned}
\mu_{21} &= \mathsf{output}(\mathsf{DR},i_k,\mathsf{GS}) \diamond \mathsf{GS.DR.Buffer} \\
\mu_{22} &= (m_1)^\star \diamond m_2 \diamond \mathsf{GS}
\end{aligned}$$

In the lock-step variant we do not have the problem of the potentially unbounded buffer, as we do in the bundle variant. The reason is that in this variant the buffer never contains more than one intersection.

Property 8 *Each solution that contains μ_{19} and μ_{20} is followed by a solution that contains μ_{21} and μ_{22}.*

Once GS has buffered the intersection, DR is ready to remove it from the buffer.

$$\begin{aligned}\mu_{23} &= \mathsf{input(GS,}\mathit{intersection}\mathsf{,DR)} \\ &\qquad \diamond (\mathsf{request(DA,}\mathit{doctrine}\mathsf{,DR)} + \mathsf{request(TS,}\mathit{track}\mathsf{,DR)}) \\ &\qquad \diamond \mathsf{output(DS,}\mathit{action}\mathsf{,DR)} \diamond (m_5)^\star \diamond m_6 \diamond \mathsf{DR} \\ \mu_{24} &= \mathsf{output(DR,}i_k\mathsf{,GS)} \diamond \mathsf{GS.DR.Buffer}\end{aligned}$$

Property 9 *Each solution that contains μ_{21} and μ_{22} is followed by a solution that contains μ_{23} and μ_{24}.*

DR then consumes the buffered intersection, leading to a state in which it asks for a doctrine and for a track. After that the buffer becomes empty again.

$$\begin{aligned}\mu_{25} &= (\mathsf{request(DA,}\mathit{doctrine}\mathsf{,DR)} + \mathsf{request(TS,}\mathit{track}\mathsf{,DR)}) \\ &\qquad \diamond \mathsf{output(DS,}\mathit{action}\mathsf{,DR)} \diamond (m_5)^\star \diamond m_6 \diamond \mathsf{DR} \\ \mu_{26} &= \mathsf{GS.DR.Buffer}\end{aligned}$$

Property 10 *Each solution that contains μ_{23} and μ_{24} is followed by a solution that contains μ_{25} and μ_{26}.*

In order to get a doctrine and a track from DA and TS, respectively, DR has a two choices. It can first ask for a doctrine and then for a track, or vice versa. Let us first consider the former case, which is expressed by the following molecules.

$$\begin{aligned}\mu_{27} &= \mathsf{input(DA,}\mathit{doctrine}\mathsf{,DR)} \diamond \mathsf{request(TS,}\mathit{track}\mathsf{,DR)} \\ &\qquad \diamond \mathsf{output(DS,}\mathit{action}\mathsf{,DR)} \diamond (m_5)^\star \diamond m_6 \diamond \mathsf{DR} \\ \mu_{28} &= \mathsf{output(DR,}d_i\mathsf{,DA)} \diamond m_7 \diamond \mathsf{DA} \\ \mu_{29} &= \mathsf{token(DA,DR)} \\ \mu_{30} &= \mathsf{current(}d_i\mathsf{)} \diamond \mathsf{DA}.\end{aligned}$$

A solution that contains the four molecules μ_{27}, μ_{28}, μ_{29}, and μ_{30} is followed by a state in which DR asks for a track.

$$\begin{aligned}\mu_{31} &= \mathsf{input(TS,}\mathit{track}\mathsf{,GS)} \\ &\qquad \diamond \mathsf{output(DS,}\mathit{action}\mathsf{,DR)} \diamond (m_5)^\star \diamond m_6 \diamond \mathsf{DR} \\ \mu_{32} &= \mathsf{output(DR,}t_j\mathsf{,TS)} \diamond m_8 \diamond \mathsf{TS} \\ \mu_{33} &= \mathsf{token(TS,DR)} \\ \mu_{34} &= \mathsf{current(}t_j\mathsf{)} \diamond \mathsf{TS} \\ \mu_{35} &= \mathsf{spent(DA,TS)}\end{aligned}$$

Similar reasonings hold for the symmetric case, when DR first asks for a track from TS and then a doctrine from DA.

$$\begin{aligned}\mu_{36} &= \mathsf{input(TS,}\mathit{track}\mathsf{,DR)} \diamond \mathsf{request(DA,}\mathit{doctrine}\mathsf{,DR)} \\ &\qquad \diamond \mathsf{output(DS,}\mathit{action}\mathsf{,DR)} \diamond (m_5)^\star \diamond m_6 \diamond \mathsf{DR} \\ \mu_{37} &= \mathsf{output(DR,}t_j\mathsf{,TS)} \diamond m_8 \diamond \mathsf{TS} \\ \mu_{38} &= \mathsf{token(TS,DR)} \\ \mu_{39} &= \mathsf{current(}t_j\mathsf{)} \diamond \mathsf{TS}\end{aligned}$$

This is followed by a state in which DR asks for a doctrine from DA.

$$\begin{aligned}
\mu_{40} &= \mathsf{input}(\mathsf{DA},\mathit{doctrine},\mathsf{DR}) \\
&\qquad \diamond \mathsf{output}(\mathsf{DS},\mathit{action},\mathsf{DR}) \diamond (m_5)^\star \diamond m_6 \diamond \mathsf{DR} \\
\mu_{41} &= \mathsf{output}(\mathsf{DR},d_i,\mathsf{DA}) \diamond m_3 \diamond \mathsf{DA} \\
\mu_{42} &= \mathsf{token}(\mathsf{DA},\mathsf{DR}), \\
\mu_{43} &= \mathsf{current}(d_i) \diamond \mathsf{DA} \\
\mu_{44} &= \mathsf{spent}(\mathsf{TS},\mathsf{DR})
\end{aligned}$$

Property 11 *Each solution that contains μ_{25} and μ_{26} is followed by either a solution that contains μ_{27}, μ_{28}, μ_{29}, and μ_{30} or by a solution that contains μ_{36}, μ_{37}, μ_{38}, and μ_{39}. Each solution that contains μ_{27}, μ_{28}, μ_{29}, and μ_{30} is followed by a solution that contains μ_{31}, μ_{32}, μ_{33}, μ_{34}, and μ_{35}. Each solution that contains μ_{36}, μ_{37}, μ_{38}, and μ_{39} is followed by a solution that contains μ_{40}, μ_{41}, μ_{42}, μ_{43}, and μ_{44}.*

Then there is a state in which DR is ready to output an action and DS is ready to input that action.

$$\begin{aligned}
\mu_{45} &= \mathsf{output}(\mathsf{DS},a_l,\mathsf{DR}) \diamond (m_5)^\star \diamond m_6 \diamond \mathsf{DR} \\
\mu_{46} &= \mathsf{input}(\mathsf{DR},\mathit{action},\mathsf{DS}) \\
&\qquad \diamond (\mathsf{request}(\mathsf{DV},\mathit{map},\mathsf{DS}) + \mathsf{request}(\mathsf{DA},\mathit{doctrine},\mathsf{DS}) + \mathsf{request}(\mathsf{TS},\mathit{track},\mathsf{DS})) \\
&\qquad \diamond (m_9)^\star \diamond m_{10} \diamond \mathsf{DS}
\end{aligned}$$

Assumption 3 *Again we can assume that the action computed by DR in molecule μ_{45} is computed from the intersection, doctrine, and track just received, namely $a_l = DR(d_i, t_j, i_k)$.*

Property 12 *Each solution that contains μ_{31}, μ_{32}, μ_{33}, μ_{34}, and μ_{35} or μ_{40}, μ_{41}, μ_{42}, μ_{43}, and μ_{44} is followed by a solution that contains μ_{45} and μ_{46}.*

The action is then actually received by DS.

$$\begin{aligned}
\mu_{47} &= (m_5)^\star \diamond m_6 \diamond \mathsf{DR} \\
\mu_{48} &= (\mathsf{request}(\mathsf{DV},\mathit{map},\mathsf{DS}) + \mathsf{request}(\mathsf{DA},\mathit{doctrine},\mathsf{DS}) + \mathsf{request}(\mathsf{TS},\mathit{track},\mathsf{DS})) \\
&\qquad \diamond (m_9)^\star \diamond m_{10} \diamond \mathsf{DS}
\end{aligned}$$

Property 13 *Each solution that contains μ_{45} and μ_{46} is followed by a solution that contains μ_{47} and μ_{48}.*

At this point DS is ready to ask for a map from DV, a doctrine from DA, and a track from TS. The choice is nondeterministic. We could go further to show that the map computed by DV uses the same doctrine and track used by DR to compute the action. This, however, follows similar reasonings as above.

Finally, we have that the lock-step architecture must satisfy the following formula.

$$
\begin{aligned}
&I(\ \mu_1 \wedge \mu_2 \wedge\ \mu_3 \wedge \mu_4,\ \mu_5 \wedge \mu_6 \wedge \mu_7 \wedge \mu_8 \wedge \mu_9\ ) \wedge \\
&I(\ \mu_{10} \wedge \mu_{11} \wedge\ \mu_{12} \wedge \mu_{13},\ \mu_{14} \wedge \mu_{15} \wedge \mu_{16} \wedge \mu_{17} \wedge \mu_{18}\ ) \wedge \\
&I(\ (\ \mu_5 \wedge \mu_6 \wedge \mu_7 \wedge \mu_8 \wedge \mu_9)\ \vee\ (\mu_{14} \wedge \mu_{15} \wedge \mu_{16} \wedge \mu_{17} \wedge \mu_{18}),\ \mu_{19} \wedge \mu_{20}\ ) \wedge \\
&I(\ \mu_{19} \wedge \mu_{20},\ \mu_{21} \wedge \mu_{22}\ ) \wedge \\
&I(\ \mu_{21} \wedge \mu_{22},\ \mu_{22} \wedge \mu_{23}\ ) \wedge \\
&I(\ \mu_{23} \wedge \mu_{24},\ \mu_{25} \wedge \mu_{26}\ ) \wedge \\
&I(\ \mu_{25} \wedge \mu_{26},\ (\mu_{27} \wedge \mu_{28} \wedge \mu_{29} \wedge \mu_{30})\ \vee\ (\mu_{36} \wedge \mu_{37} \wedge \mu_{38} \wedge \mu_{39})\ ) \wedge \\
&I(\ \mu_{27} \wedge \mu_{28} \wedge \mu_{29} \wedge \mu_{30},\ \mu_{31} \wedge \mu_{32} \wedge \mu_{33} \wedge \mu_{34} \wedge \mu_{35}\ ) \wedge \\
&I(\ \mu_{36} \wedge \mu_{37} \wedge \mu_{38} \wedge \mu_{39},\ \mu_{40} \wedge \mu_{41} \wedge \mu_{42} \wedge \mu_{43} \wedge \mu_{44})\ ) \wedge \\
&I(\ (\mu_{31} \wedge \mu_{32} \wedge \mu_{33} \wedge \mu_{34} \wedge \mu_{35})\ \vee\ (\mu_{40} \wedge \mu_{41} \wedge \mu_{42} \wedge \mu_{43} \wedge \mu_{44}),\ \mu_{45} \wedge \mu_{46})\ ) \wedge \\
&I(\ \mu_{45} \wedge \mu_{46},\ \mu_{47} \wedge \mu_{48}\ )
\end{aligned}
$$

4.3 Validating Identifiers

GS asks for a doctrine and a track from DA and TS, respectively. Since the computation is driven by GS, this latter process fixes two identifiers when rule T_4 is applied: one for the doctrine and another for the track. We make the following assumption for this architecture.

Assumption 4 *Components DA and TS provide the same data given the same identifiers.*

Once GS has fixed the identifiers for the doctrine and for the track it can ask for a doctrine and track from DA and TS, respectively. As above, this is done in two possible ways: GS asks for a doctrine from DA and then asks for a track from TS, or vice versa.

GS asks for a doctrine from DA when it is ready to output a doctrine identifier to DA and then it can input the associated doctrine from DA, and DA is ready to input the doctrine identifier from GS and then it can output the associated doctrine to DA.

$$
\begin{aligned}
\mu_1 = {} & \mathsf{output}(\mathsf{DA},\mathit{id}_i,\mathsf{GS}) \diamond \mathsf{input}(\mathsf{DA},\mathit{doctrine},\mathsf{GS}) \diamond \mathsf{request}(\mathsf{TS},\{\mathit{id}_j,\mathit{track}\},\mathsf{GS}) \\
& \diamond \mathsf{buffer}(\mathsf{DR},\{\mathit{id}_i,\mathit{id}_j,\mathit{intersection}\},\mathsf{GS}) \diamond (m_1)^\star \diamond m_2 \diamond \mathsf{GS} \\
\mu_2 = {} & \mathsf{input}(\mathsf{GS},\mathit{identifier}_1,\mathsf{DA}) \diamond \mathsf{output}(\mathsf{GS},\mathit{doctrine},\mathsf{DA}) \diamond m_3 \diamond \mathsf{DA}
\end{aligned}
$$

A solution that contains molecules μ_1 and μ_2 is followed by a solution in which the doctrine associated with the identifier id_i is actually received by GS. This involves the application of rule T_8 twice. In the first application, GS communicates to DA the doctrine identifier and in the second application DA communicates to GS the doctrine associated with that identifier.

Once GS has obtained a doctrine from DA it can ask for a track from TS. The protocol is the same. First GS communicates to TS a track identifier and then it waits for the track itself.

$$
\begin{aligned}
\mu_3 = {} & \mathsf{output}(\mathsf{TS},\mathit{id}_j,\mathsf{GS}) \diamond \mathsf{input}(\mathsf{TS},\mathit{track},\mathsf{GS}) \\
& \diamond \mathsf{buffer}(\mathsf{DR},\{\mathit{id}_i,\mathit{id}_j,\mathit{intersection}\},\mathsf{GS}) \diamond (m_1)^\star \diamond m_2 \diamond \mathsf{GS} \\
\mu_4 = {} & \mathsf{input}(\mathsf{GS},\mathit{identifier}_2,\mathsf{TS}) \diamond \mathsf{output}(\mathsf{GS},\mathit{track},\mathsf{TS}) \diamond m_4 \diamond \mathsf{TS}
\end{aligned}
$$

If GS first asks for a track and then for a doctrine, the situation is symmetric.

$$
\begin{aligned}
\mu_5 = {} & \mathsf{output}(\mathsf{TS},\mathit{id}_j,\mathsf{GS}) \diamond \mathsf{input}(\mathsf{TS},\mathit{track},\mathsf{GS}) \diamond \mathsf{request}(\mathsf{DA},\{\mathit{id}_i,\mathit{doctrine}\},\mathsf{GS}) \\
& \diamond \mathsf{buffer}(\mathsf{DR},\{\mathit{id}_i,\mathit{id}_j,\mathit{intersection}\},\mathsf{GS}) \diamond (m_1)^\star \diamond m_2 \diamond \mathsf{GS} \\
\mu_6 = {} & \mathsf{input}(\mathsf{GS},\mathit{identifier}_2,\mathsf{TS}) \diamond \mathsf{output}(\mathsf{GS},\mathit{track},\mathsf{TS}) \diamond m_4 \diamond \mathsf{TS}
\end{aligned}
$$

This is followed by a state in which the track is actually received and GS is ready to input a doctrine.

$$\begin{aligned}\mu_7 =& \text{output(DA,}\textit{id}_i\text{,GS)} \diamond \text{input(DA,}\textit{doctrine}\text{,GS)}\\ & \diamond \text{buffer(DR,}\{\textit{id}_i\text{,}\textit{id}_j\text{,}\textit{intersection}\}\text{,GS)} \diamond (m_1)^\star \diamond m_2 \diamond \text{GS}\\ \mu_8 =& \text{input(GS,}\textit{identifier}_1\text{,DA)} \diamond \text{output(GS,}\textit{doctrine}\text{,DA)} \diamond m_3 \diamond \text{DA}\end{aligned}$$

Property 14 *Each solution that contains μ_1 and μ_2 is followed by a solution that contains μ_3 and μ_4. Each solution that contains μ_5 and μ_6 is followed by a solution that contains μ_7 and μ_8.*

Once GS has obtained both a doctrine and a track using their identifiers, it is ready to buffer an intersection. This is expressed by a solution that contains molecule μ_9.

$$\mu_9 = \text{buffer(DR,}\{\textit{id}_i\text{,}\textit{id}_j\text{,}i_k\}\text{,GS)} \diamond (m_1)^\star \diamond m_2 \diamond \text{GS}$$

Property 15 *Each solution contains μ_3 and μ_4 or μ_7 and μ_8 is followed by a solution that contains μ_9.*

Then the intersection is actually buffered. This is expressed by a solution that contains molecules μ_{10} and μ_{11}.

$$\begin{aligned}\mu_{10} &= \text{output(DR,}\{\textit{id}_i\text{,}\textit{id}_j\text{,}i_k\}\text{,GS)} \diamond \ldots \diamond \text{GS.DR.Buffer}\\ \mu_{11} &= (m_1)^\star \diamond m_2 \diamond \text{GS}\end{aligned}$$

Property 16 *Each solution that contains μ_9 is followed by a solution that contains μ_{10} and μ_{11}.*

The intersection buffered by GS will be used by DR to compute an action. This is possible when DR is ready to remove an intersection from the buffer and the buffer is not empty.

$$\begin{aligned}\mu_{12} =& \text{output(DR,}\{\textit{id}_i\text{,}\textit{id}_j\text{,}i_k\}\text{,GS)} \diamond \ldots \diamond \text{GS.DR.Buffer}\\ \mu_{13} =& \text{input(GS,}\{\textit{identifier}_1\text{,}\textit{identifier}_2\text{,}\textit{intersection}\}\text{,DR)}\\ & \diamond (\text{request(DA,}\{\textit{identifier}_1\text{,}\textit{doctrine}\}\text{,DR)} + \text{request(TS,}\{\textit{identifier}_2\text{,}\textit{track}\}\text{,DR)})\\ & \diamond \text{output(DS,}\{\textit{identifier}_1\text{,}\textit{identifier}_2\text{,}\textit{action}\}\text{,DR)} \diamond (m_5)^\star \diamond m_6 \diamond \text{DR}\end{aligned}$$

Property 17 *Each solution that contains μ_{10} and μ_{11} is followed by a solution that contains μ_{12} and μ_{13}.*

From the doctrine and track identifiers, DR tries to get the associated doctrine and track in order to compute an action. As usual it can first receive the doctrine and then the track, or vice versa. Assume that DR first asks for the doctrine and then for the track.

$$\begin{aligned}\mu_{14} =& \text{output(DA,}\textit{id}_i\text{,DR)} \diamond \text{input(DA,}\textit{doctrine}\text{,DR)} \diamond \text{request(TS,}\{\textit{id}_j\text{,}\textit{track}\}\text{,DR)}\\ & \diamond \text{output(DS,}\{\textit{id}_i\text{,}\textit{id}_j\text{,}\textit{action}\}\text{,DR)} \diamond (m_7)^\star \diamond m_8 \diamond \text{DR}\\ \mu_{15} =& \text{input(DR,}\textit{identifier}_1\text{,DA)} \diamond \text{output(DR,}\textit{doctrine}\text{,DA)} \diamond m_9 \diamond \text{DA}\end{aligned}$$

A solution that contains molecules μ_{14} and μ_{15} is followed by a solution in which DR asks for the track associated with a given identifier.

$$\begin{aligned}\mu_{16} =& \text{output(TS,}\textit{id}_j\text{,DR)} \diamond \text{input(TS,}\textit{track}\text{,DR)}\\ & \diamond \text{output(DS,}\{\textit{id}_i\text{,}\textit{id}_j\text{,}\textit{action}\}\text{,DR)} \diamond (m_7)^\star \diamond m_8 \diamond \text{DR}\\ \mu_{17} =& \text{input(DR,}\textit{identifier}_2\text{,TS)} \diamond \text{output(DR,}\textit{track}\text{,TS)} \diamond m_{10} \diamond \text{TS}\end{aligned}$$

Symmetrically, DR can first ask for a track and then a doctrine.

$$\begin{aligned}\mu_{18} &= \mathsf{output}(\mathsf{TS},\mathit{id}_j,\mathsf{DR}) \diamond \mathsf{input}(\mathsf{TS},\mathit{track},\mathsf{DR}) \diamond \mathsf{request}(\mathsf{DA},\{\mathit{id}_i,\mathit{doctrine}\},\mathsf{DR}) \\ &\qquad \diamond \mathsf{output}(\mathsf{DS},\{\mathit{id}_i,\mathit{id}_j,\mathit{action}\},\mathsf{DR}) \diamond (m_7)^\star \diamond m_8 \diamond \mathsf{DR} \\ \mu_{19} &= \mathsf{input}(\mathsf{DR},\mathit{identifier}_2,\mathsf{TS}) \diamond \mathsf{output}(\mathsf{DR},\mathit{track},\mathsf{TS}) \diamond m_{10} \diamond \mathsf{TS}\end{aligned}$$

A solution that contains molecules μ_{18} and μ_{19} is followed by a solution in which DR asks for the doctrine associated with a given identifier.

$$\begin{aligned}\mu_{20} &= \mathsf{output}(\mathsf{DA},\mathit{id}_i,\mathsf{DR}) \diamond \mathsf{input}(\mathsf{DA},\mathit{doctrine},\mathsf{DR}) \\ &\qquad \diamond \mathsf{output}(\mathsf{DS},\{\mathit{id}_i,\mathit{id}_j,\mathit{action}\},\mathsf{DR}) \diamond (m_7)^\star \diamond m_8 \diamond \mathsf{DR} \\ \mu_{21} &= \mathsf{input}(\mathsf{DR},\mathit{identifier}_1,\mathsf{DA}) \diamond \mathsf{output}(\mathsf{DR},\mathit{doctrine},\mathsf{DA}) \diamond m_9 \diamond \mathsf{DA}\end{aligned}$$

Property 18 *Each solution that contains μ_{14} and μ_{15} is followed by a solution that contains μ_{16} and μ_{17}. Each solution that contains μ_{18} and μ_{19} is followed by a solution that contains μ_{20} and μ_{21}.*

Then DR computes an action and sends it to DS.

$$\begin{aligned}\mu_{22} &= \mathsf{output}(\mathsf{DS},\{\mathit{id}_i,\mathit{id}_j,\mathit{a}_l\},\mathsf{DR}) \diamond (m_7)^\star \diamond m_8 \diamond \mathsf{DR} \\ \mu_{23} &= \mathsf{input}(\mathsf{DR},\{\mathit{identifier}_1,\mathit{identifier}_2,\mathit{action}\},\mathsf{DS}) \\ &\qquad \diamond (\mathsf{request}(\mathsf{DV},\{\{\mathit{identifier}_1,\mathit{identifier}_2\},\mathit{map}\},\mathsf{DS}) \\ &\qquad\quad + \mathsf{request}(\mathsf{DA},\{\mathit{identifier}_1,\mathit{doctrine}\},\mathsf{DS}) + \mathsf{request}(\mathsf{TS},\{\mathit{identifier}_2,\mathit{track}\},\mathsf{DS})) \\ &\qquad \diamond (m_{11})^\star \diamond m_{12} \diamond \mathsf{DA}\end{aligned}$$

Property 19 *Each solution that contains μ_{16} and μ_{17} or μ_{20} and μ_{21} is followed by a solution that contains μ_{22} and μ_{23}.*

The action is then sent to DS.

$$\begin{aligned}\mu_{24} &= (m_7)^\star \diamond m_8 \diamond \mathsf{DR} \\ \mu_{25} &= (\mathsf{request}(\mathsf{DV},\{\{\mathit{id}_i,\mathit{id}_j\},\mathit{map}\},\mathsf{DS}) \\ &\qquad\qquad + \mathsf{request}(\mathsf{DA},\{\mathit{id}_i,\mathit{doctrine}\},\mathsf{DS}) + \mathsf{request}(\mathsf{TS},\{\mathit{id}_j,\mathit{track}\},\mathsf{DS})) \\ &\qquad \diamond (m_{11})^\star \diamond m_{12} \diamond \mathsf{DA}\end{aligned}$$

Property 20 *Each solution that contains μ_{22} and μ_{23} is followed by a solution that contains μ_{24} and μ_{25}.*

According to the received doctrine identifier and track identifier, DS asks for a map, a doctrine, and a track. This is enough to depict the map. Again, this is omitted because it follows similar reasoning as above.

The identifier architecture must satisfy the following formula.

$$
\begin{aligned}
&I(\, \mu_1 \wedge \mu_2,\ \mu_3 \wedge \mu_4 \,) \wedge \\
&I(\, \mu_5 \wedge \mu_6,\ \mu_7 \wedge \mu_8 \,) \wedge \\
&I(\, (\mu_3 \wedge \mu_4) \vee (\mu_7 \wedge \mu_8),\ \mu_9 \,) \wedge \\
&I(\, \mu_9,\ \mu_{10} \wedge \mu_{11} \,) \wedge \\
&J(\, \mu_{10} \wedge \mu_{11},\ \mu_{12} \wedge \mu_{13} \,) \wedge \\
&I(\, \mu_{12} \wedge \mu_{13},\ (\mu_{14} \wedge \mu_{15}) \vee (\mu_{18} \wedge \mu_{19}) \,) \\
&I(\, \mu_{14} \wedge \mu_{15},\ \mu_{16} \wedge \mu_{17} \,) \\
&I(\, \mu_{18} \wedge \mu_{19},\ \mu_{20} \wedge \mu_{21} \,) \\
&I(\, (\mu_{16} \wedge \mu_{17}) \vee (\mu_{20} \wedge \mu_{21}),\ \mu_{22} \wedge \mu_{23} \,) \\
&I(\, \mu_{22} \wedge \mu_{23},\ \mu_{24} \wedge \mu_{25} \,)
\end{aligned}
$$

5 Conclusion

The relationship between architecture and design has been a long-standing subject of discussion. What has emerged is an understanding that "architecture" should refer to high levels of design that can be related to lower levels of design through suitable refinement steps. This seems reasonable, since software architectures represent the first system abstraction in which a (high level) description of the system implementation is provided, and that its structure and behavior should be prescriptive with respect to the implementation.

The relationship between architecture and requirements is, by contrast, less clear. On one side there is the awareness that requirements and architectures should be related [24]. On the other there is the difficulty of relating very different concerns, modeled in different ways.

In this paper we tackled an instance of this problem by showing how a requirement on the data coherency of a system, modeled in a functional way, can be related to three variants of a software architecture, modeled in an operational, state-based style. At a very basic level, our approach is a kind of model checking, since it considers the requirement as a property that has to be related to a model of the implementation. This is a straightforward approach and there is no new insight in and of itself. What is novel in our approach is the way we build the formula that represents the relationship between the requirement and the architecture. In particular, we translate the requirement into a form that constraints the states of the architectural model. This is the most difficult part of our work, and the way we carried it out follows a logical path that allows us to easily prove that the formula is satisfied by the model. In other words, in building the formula we also performed the bulk of the proof. In more formal terms, we performed a proof by construction.

This suggests a future line of research in the direction of (semi-)automating the approach presented here. The idea is to formalize our methodology so that, starting from a functional requirement and from a basic set of mappings (e.g., basic functionalities on components, inputs/outputs on states, etc.) it can be possible to automatically build sub-formulae by model checking. Then the formulae can be composed together as suggested in our approach by using suitable logical and temporal operators. This would ease the task of relating very different models and, thus, allow early validation of software architectures with respect to complex functional constraints.

References

[1] G.D. Abowd, R. Allen, and D. Garlan. Formalizing Style to Understand Descriptions of Software Architecture. *ACM Transactions on Software Engineering and Methodology*, 4(4):319–364, October 1995.

[2] R. Allen and D. Garlan. A Case Study in Architectural Modeling: The AEGIS System. In *Proceedings of the 8th International Workshop on Software Specification and Design*, pages 6–15. IEEE Computer Society, March 1996.

[3] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.

[4] F. Aquilani, S. Balsamo, and P. Inverardi. Performance Analysis at the Software Architectural Design Level. *Performance Evaluation*, July 2001.

[5] L. Bass, P. Clements, and R. Kazman, editors. *Software Architecture in Practice*. Addison-Wesley, Reading, Massachusetts, 1998.

[6] G. Berry and G. Boudol. The Chemical Abstract Machine. *Theoretical Computer Science*, 96:217–248, 1992.

[7] A. Bertolino, F. Corradini, P. Inverardi, and H. Muccini. Deriving Test Plans from Architectural Descriptions. In *Proceedings of the 2000 International Conference on Software Engineering*, pages 220–229. Association for Computer Machinery, June 2000.

[8] A. Bertolino, P. Inverardi, and H. Muccini. An Explorative Journey from Architectural Tests Definition Down to Code Test Execution. In *Proceedings of the 2001 International Conference on Software Engineering*, pages 211–220. Association for Computer Machinery, May 2001.

[9] D. Compare, P. Inverardi, and A.L. Wolf. Uncovering Architectural Mismatch in Component Behavior. *Science of Computer Programming*, 33(2):101–131, February 1999.

[10] E.A. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B: Formal Models and Semantics*, pages 996–1072. Elsevier, 1990.

[11] E.A. Emerson and J.Y. Halpern. “Sometimes” and “Not Never” Revisited: On Branching versus Linear Time Temporal Logic. *Journal of the ACM*, 33(1):151–178, January 1986.

[12] D. Garlan, W. Tichy, and F. Paulisch. Summary of the Dagstuhl Workshop on Software Architecture. *SIGSOFT Software Engineering Notes*, 20(3):63–83, July 1995.

[13] C. Hofmeister, R.L. Nord, and D. Soni. *Applied Software Architecture*. Addison-Wesley, Reading, Massachusetts, 2000.

[14] P. Inverardi and A.L. Wolf. Formal Specification and Analysis of Software Architectures using the Chemical Abstract Machine Model. *IEEE Transactions on Software Engineering*, 21(4):373–386, April 1995.

[15] P. Inverardi, A.L. Wolf, and D. Yankelevich. Static Checking of System Behaviors Using Derived Component Assumptions. *ACM Transactions on Software Engineering and Methodology*, 9(3):239–272, July 2000.

[16] J. Kramer and J. Castro, editors. *Proceedings of the First International Workshop from Software Requirements to Architectures (STRAW '01).* IEEE Computer Society, May 2001.

[17] D.C. Luckham, J.J. Kenney, L.M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, April 1995.

[18] D. Le Métayer. Describing Software Architecture Styles Using Graph Grammars. *IEEE Transactions on Software Engineering*, 24(7):521–533, July 1998.

[19] G. Naumovich, G.S. Avrunin, L.A. Clarke, and L.J. Osterweil. Applying Static Analysis to Software Architectures. In *Proceedings of the Sixth European Software Engineering Conference Held Jointly with the 5th ACM SIGSOFT Symposium on Foundations of Software Engineering*, number 1301 in Lecture Notes in Computer Science, pages 77–93. Springer-Verlag, 1997.

[20] B. Nuseibeh. Weaving Together Requirements and Architecture. *Computer*, 34(3):115–117, March 2001.

[21] J.A. Stafford and A.L. Wolf. Annotating Components to Support Component-Based Static Analyses of Software Systems. In *Proceedings Grace Hopper Celebration of Women in Computing 2000*, September 2000.

[22] J.A. Stafford and A.L. Wolf. Architecture-Level Dependence Analysis for Software Systems. *International Journal of Software Engineering and Knowledge Engineering*, 11(4):431–452, August 2001.

[23] J.A. Stafford and A.L. Wolf. Software Architecture. In G.T. Heineman and W.T. Councill, editors, *Component-Based Software Engineering: Putting the Pieces Together*, pages 371–387. Addison-Wesley, Reading, Massachusetts, 2001.

[24] A. van Lamsweerde. Requirements Engineering in the Year 00: A Research Perspective. In *Proceedings of the 2000 International Conference on Software Engineering*, pages 5–19. Association for Computer Machinery, June 2000.

[25] M. Wermelinger. Towards a Chemical Model for Software Architecture Reconfiguration. In *Proceedings of the 4th International Conference on Configurable Distributed Systems*, pages 111–118. IEEE Computer Society, May 1998.

A Background

This appendix provides a brief review of the CHAM formalism [6], Kripke structures, and the CTL logic [10, 11].

A.1 The CHAM Model

A CHAM is specified by defining *molecules* $m_1, m_2, \ldots$ defined as terms of a syntactic algebra that derive from a set of constants and a set of operations, and *solutions* $S_0, S_1, \ldots$ of molecules. Molecules constitute the basic elements of a CHAM, while solutions are multisets of molecules interpreted as defining the *states* of a CHAM. One of these solutions is identified as the *initial solution*, which is used to represent the initial state of the CHAM. A CHAM specification contains *transformation rules* $T_1, T_2, \ldots$ (of the form $m_1, m_2, \ldots, m_i \longrightarrow m_{i+1}, m_{i+2}, \ldots, m_{i+j}$) that define a transformation relation from solutions to solutions, $S_k \longrightarrow S_l$, dictating the way solutions can evolve (i.e., states can change) in the CHAM. The transformation relation is given via an inference rule of the form $S_1 \longrightarrow S_2$ implies $S_1 \uplus S_3 \longrightarrow S_2 \uplus S_3$.

For a given solution, a CHAM can apply as many rules as possible, provided that their premises do not conflict (that is, no molecule is involved in more than one rule). This permits the modeling of parallel behaviors by performing parallel transformations. When more than one rule can apply to the same (set of) molecules, then a nondeterministic choice is made as to which transformation to apply.

As mentioned in Section 3, the CHAM description of a software architecture [14, 15] consists of a syntactic description of the static components of the architecture (the molecules), a solution representing the initial state of the system in terms of the initial states of its architectural components (the initial solution), and of a set of transformation rules that describe how the system dynamically evolves.

A.2 Kripke Structures

The CHAM formalism accommodates a variety of architectural analysis techniques [9, 14]. Thus, depending on the property of interest, one can choose the most adequate technique. In particular, one can either exploit the algebraic and equational nature of CHAM, or one can take advantage of its operational flavor to derive a transition system or a Kripke structure and then reason at this level of abstraction.

Let us show how to derive a Kripke structure from a CHAM specification. In essence, we derive the Kripke structure from the transformation rules, which are the basis for the operational description of the CHAM.

Definition 1 *(Operational semantics induced by $\mathcal{T}$) Let $\mathcal{T}$ be the set of transformation rules of a CHAM C. Then $\mathcal{T}$ defines a relation $\rightarrow_{\mathcal{T}} \subseteq S \times S$, where S is the set of solutions. The relation is the least relation satisfying the rules.*

Definition 2 *(Derivative) Given a set of transformation rules $\mathcal{T}$, a $\mathcal{T}$-derivation from a solution S to a solution S_n is a sequence $\{S_i, 1 \leq i \leq n, n > 1\}$ such that $S = S_1$ and for any $1 \leq i \leq n-1$, $S_i \rightarrow_{\mathcal{T}} S_{i+1}$. A solution S is called a $\mathcal{T}$-*derivative *of S' if a $\mathcal{T}$-derivation exists from S' to S. The set of derivatives of S is denoted by $D_{\mathcal{T}}(S)$, while $M_{\mathcal{T}}(S)$ denotes the set of molecules within solutions in $D_{\mathcal{T}}(S)$.*

Definition 3 *(Kripke Structure) A* Kripke structure *(or KS) is a 5-tuple* $\mathcal{K} = (\mathcal{S}, \mathcal{AP}, \mathcal{L}, \rightarrow, s_0)$, *where*

- $\mathcal{S}$ *is a set of states;*
- $\mathcal{AP}$ *is a non-empty set of* atomic proposition names *ranged over by* $p, p_1, \ldots$*;*
- $\mathcal{L} : \mathcal{S} \rightarrow 2^{\mathcal{AP}}$ *is a function that assigns to each state a set of atomic propositions true in that state;*
- $\rightarrow \subseteq \mathcal{S} \times \mathcal{S}$ *is the* transition relation*; and*
- s_0 *is the initial state.*

We can now state how, given a CHAM and a solution, we can derive a Kripke structure that represents the whole set of possible derivations. If the number of derivable solutions is finite, then the Kripke structure is also finite.

Definition 4 *(Kripke Structure corresponding to a solution) Given a solution* S *and a set of transformation rules* $\mathcal{T}$, $\mathcal{T}(S)$ *is the Kripke structure* $(D_{\mathcal{T}}(S) \cup \{S\}, M_{\mathcal{T}}(S), \mathcal{L}, \rightarrow_{\mathcal{T}}, S)$*, where for every* $S' \in D_{\mathcal{T}}(S) \cup \{S\}$, $\mathcal{L}(S')$ *is the set of molecules in* S' *and* $\rightarrow_{\mathcal{T}}$ *is the relation defined by* $\mathcal{T}$*.*

A.3 The CTL Logic

The CTL logic is suitable to express properties of reactive systems defined by means of transition systems or Kripke structures. Before defining the syntax and semantics of CTL operators, let us introduce some notation and definitions.

Let $\mathcal{K} = (\mathcal{S}, \mathcal{AP}, \mathcal{L}, \rightarrow, s_0)$ be a Kripke structure. A *path* is defined as follows.

- σ is a path from $r_0 \in \mathcal{S}$ if either $\sigma = r_0$ (the empty path from r_0) or σ is a (possibly infinite) sequence $(r_0, r_1)(r_1, r_2) \ldots$ such that $(r_i, r_{i+1}) \in \rightarrow$ for each $i \geq 0$.
- A path σ is called *maximal* if either it is infinite or it is finite and its last state r has no successor states. The set of maximal paths from r_0 is denoted $\Pi(r_0)$.
- When $\sigma = \rho\eta$, we say that ρ is a prefix of σ.
- If σ is infinite, then $|\sigma| = \omega$.
  If $\sigma = r_0$, then $|\sigma| = 0$.
  If $\sigma = (r_0, r_1)(r_1, r_2) \ldots (r_n, r_{n+1})$, $n \geq 0$, then $|\sigma| = n + 1$. Moreover, we denote the ith state in the sequence, r_i, by $\sigma(i)$.

The syntax of CTL is defined by the state and path formulae generated by the following grammar.

$$\begin{array}{lcl} \phi & ::= & p \mid \phi \wedge \phi \mid \neg\phi \mid E\gamma \mid A\gamma \\ \gamma & ::= & X\phi \mid F\phi \mid \phi U \phi \end{array}$$

where ϕ ranges over state formulae, γ ranges over path formulae, p ranges over $\mathcal{AP}$, E (for some path) and A (for all paths) are path quantifiers, and X, U and F are *next*, *until*, and *sometimes* operators, respectively.

Intuitively, the *next* modality $X\phi$ says that the next state of the path holds formula ϕ. The *until* modality $\phi U \phi'$ says that along the path, all states satisfy ϕ, until a state that satisfies ϕ' is reached. The *sometime* modality $F\phi$ says that along the path there exists a state that satisfies ϕ.

Satisfaction of a state formula ϕ (path formula γ) by a state s (path σ), which is denoted $s \models_{\mathcal{K}} \phi$ ($\sigma \models_{\mathcal{K}} \gamma$), is given inductively by the following.

$$
\begin{array}{lll}
s \models_{\mathcal{K}} p & \text{iff} & p \in \mathcal{L}(s) \\
s \models_{\mathcal{K}} \phi \wedge \phi' & \text{iff} & s \models_{\mathcal{K}} \phi \text{ and } s \models_{\mathcal{K}} \phi' \\
s \models_{\mathcal{K}} \neg\phi & \text{iff} & \text{not } s \models_{\mathcal{K}} \phi \\
s \models_{\mathcal{K}} E\gamma & \text{iff} & \text{there exists a path } \sigma \in \Pi(s) \text{ such that } \sigma \models_{\mathcal{K}} \gamma \\
s \models_{\mathcal{K}} A\gamma & \text{iff} & \text{for all paths } \sigma \in \Pi(s),\ \sigma \models_{\mathcal{K}} \gamma \\
\sigma \models_{\mathcal{K}} F\phi & \text{iff} & \text{there exists } i \geq 0 \text{ and } \sigma(i) \models_{\mathcal{K}} \phi \\
\sigma \models_{\mathcal{K}} X\phi & \text{iff} & |\sigma| \geq 1 \text{ and } \sigma(1) \models_{\mathcal{K}} \phi \\
\sigma \models_{\mathcal{K}} \phi U \phi' & \text{iff} & \text{there exists } i \geq 0 \text{ such that } \sigma(i) \models_{\mathcal{K}} \phi', \text{ and for all} \\
 & & 1 \leq j \leq i-1,\ \sigma(j) \models_{\mathcal{K}} \phi
\end{array}
$$

B Full Specification of Variants

This appendix provides the full specifications for each of the variants described in Section 3.2.

B.1 First Variant: Identifiers

$$
\begin{aligned}
M &::= P \mid C \mid M \diamond M \mid M^\star \mid M \parallel M \\
P &::= \mathsf{EC} \mid \mathsf{DV} \mid \mathsf{DS} \mid \mathsf{DA} \mid \mathsf{TS} \mid \mathsf{GS} \mid \mathsf{DR} \\
D &::= \mathit{intersection} \mid \mathit{track} \mid \mathit{action} \mid \mathit{map} \mid \mathit{rule} \mid \\
&\quad\;\; \mathit{command} \mid \mathit{doctrine} \mid \mathit{identifier} \mid \{D'\} \\
D' &::= D \mid D, D' \\
C &::= \mathsf{input}(P,D,P) \mid \mathsf{output}(P,D,P) \mid \mathsf{connect}(P,P) \mid \mathsf{disconnect}(P,P) \mid \\
&\quad\;\; \mathsf{open}(P) \mid \mathsf{join} \mid \mathsf{closed} \mid \mathsf{request}(P,D,P) \mid \mathsf{serve}(P) \mid \\
&\quad\;\; \mathsf{buffer}(P,D,P) \mid P.P.\mathsf{Buffer} \mid C + C
\end{aligned}
$$

$$
\begin{aligned}
S_1 = \; & (\mathsf{output}(\mathsf{DA},\{\mathit{rule},\ldots\},\mathsf{EC}) + \mathsf{output}(\mathsf{DV},\{\mathit{command},\ldots\},\mathsf{EC}) \\
& \quad + \mathsf{output}(\mathsf{TS},\{\mathit{track},\ldots\},\mathsf{EC})) \diamond \mathsf{EC}, \\
& \mathsf{input}(\mathsf{EC},\{\mathit{rule},\ldots\},\mathsf{DA}) \diamond \mathsf{open}(\mathsf{DA}) \diamond \mathsf{closed} \diamond \mathsf{DA}, \\
& \mathsf{input}(\mathsf{EC},\{\mathit{track},\ldots\},\mathsf{TS}) \diamond \mathsf{open}(\mathsf{TS}) \diamond \mathsf{closed} \diamond \mathsf{TS}, \\
& \mathsf{input}(\mathsf{EC},\{\mathit{command},\ldots\},\mathsf{DV}) \diamond (\mathsf{connect}(\mathsf{DA},\mathsf{DV}) + \mathsf{connect}(\mathsf{TS},\mathsf{DV})) \\
& \quad \diamond \mathsf{open}(\mathsf{DV}) \diamond \mathsf{closed} \diamond (\mathsf{disconnect}(\mathsf{DA},\mathsf{DV}) + \mathsf{disconnect}(\mathsf{TS},\mathsf{DV})) \diamond \mathsf{DV}, \\
& (\mathsf{connect}(\mathsf{DA},\mathsf{GS}) + \mathsf{connect}(\mathsf{TS},\mathsf{GS})) \\
& \quad \diamond ((\mathsf{request}(\mathsf{DA},\{\mathit{identifier}_1,\mathit{doctrine}\},\mathsf{GS}) + \mathsf{request}(\mathsf{TS},\{\mathit{identifier}_2,\mathit{track}\},\mathsf{GS})) \\
& \quad\quad \diamond \mathsf{buffer}(\mathsf{DR},\{\mathit{identifier}_1,\mathit{identifier}_2,\mathit{intersection}\},\mathsf{GS}))^\star \\
& \quad \diamond (\mathsf{disconnect}(\mathsf{DA},\mathsf{GS}) + \mathsf{disconnect}(\mathsf{TS},\mathsf{GS})) \diamond \mathsf{GS}, \\
& (\mathsf{connect}(\mathsf{DA},\mathsf{DR}) + \mathsf{connect}(\mathsf{TS},\mathsf{DR})) \\
& \quad \diamond (\mathsf{input}(\mathsf{GS},\{\mathit{identifier}_1,\mathit{identifier}_2,\mathit{intersection}\},\mathsf{DR}) \\
& \quad\quad \diamond (\mathsf{request}(\mathsf{DA},\{\mathit{identifier}_1,\mathit{doctrine}\},\mathsf{DR}) + \mathsf{request}(\mathsf{TS},\{\mathit{identifier}_2,\mathit{track}\},\mathsf{DR})) \\
& \quad\quad \diamond \mathsf{output}(\mathsf{DS},\{\mathit{identifier}_1,\mathit{identifier}_2,\mathit{action}\},\mathsf{DR}))^\star \\
& \quad \diamond (\mathsf{disconnect}(\mathsf{DA},\mathsf{DR}) + \mathsf{disconnect}(\mathsf{TS},\mathsf{DR})) \diamond \mathsf{DR}, \\
& (\mathsf{connect}(\mathsf{DA},\mathsf{DS}) + \mathsf{connect}(\mathsf{TS},\mathsf{DS}) + \mathsf{connect}(\mathsf{DV},\mathsf{DS})) \\
& \quad \diamond (\mathsf{input}(\mathsf{DR},\{\mathit{identifier}_1,\mathit{identifier}_2,\mathit{action}\},\mathsf{DS}) \\
& \quad\quad \diamond (\mathsf{request}(\mathsf{DV},\{\{\mathit{identifier}_1,\mathit{identifier}_2\},\mathit{map}\},\mathsf{DS}) \\
& \quad\quad\quad + \mathsf{request}(\mathsf{DA},\{\mathit{identifier}_1,\mathit{doctrine}\},\mathsf{DS}) \\
& \quad\quad\quad + \mathsf{request}(\mathsf{TS},\{\mathit{identifier}_2,\mathit{track}\},\mathsf{DS})))^\star \\
& \quad \diamond (\mathsf{disconnect}(\mathsf{DA},\mathsf{DS}) + \mathsf{disconnect}(\mathsf{TS},\mathsf{DS}) + \mathsf{disconnect}(\mathsf{DV},\mathsf{DS})) \diamond \mathsf{DS}, \\
& \mathsf{GS}.\mathsf{DR}.\mathsf{Buffer}
\end{aligned}
$$

$$T_1 \equiv m_1 \parallel m_2 \longrightarrow m_1,\ m_2$$

$$T_2 \equiv (c_1 + c_2) \diamond m \longrightarrow c_1 \diamond c_2 \diamond m$$

$$T_3 \equiv (c_1 + c_2) \diamond m \longrightarrow c_2 \diamond c_1 \diamond m$$

$$T_4 \equiv (m_1)^\star \diamond m_2 \diamond \mathsf{GS} \longrightarrow m_1 \diamond (m_1)^\star \diamond m_2 \diamond \mathsf{GS}$$

$$T_5 \equiv (m_1)^\star \diamond m_2 \diamond \mathsf{GS} \longrightarrow m_2 \diamond \mathsf{GS}$$

$$T_6 \equiv (m_1)^\star \diamond m_2 \diamond \mathsf{DR},\ m_3 \diamond \mathsf{GS.DR.Buffer} \longrightarrow \\ m_1 \diamond (m_1)^\star \diamond m_2 \diamond \mathsf{DR},\ m_3 \diamond \mathsf{GS.DR.Buffer}$$

$$T_7 \equiv (m_1)^\star \diamond m_2 \diamond \mathsf{DS},\ \mathsf{output}(\mathsf{DS},d,\mathsf{DR}) \diamond m_3 \longrightarrow \\ m_1 \diamond (m_1)^\star \diamond m_2 \diamond \mathsf{DS},\ \mathsf{output}(\mathsf{DS},d,\mathsf{DR}) \diamond m_3$$

$$T_8 \equiv \mathsf{input}(p_2,d,p_1) \diamond m_1,\ \mathsf{output}(p_1,d,p_2) \diamond m_2 \longrightarrow m_1,\ m_2$$

$$T_9 \equiv \mathsf{buffer}(p_2,d,p_1) \diamond m_1,\ p_1.p_2.\mathsf{Buffer} \longrightarrow \\ m_1,\ \mathsf{output}(p_2,d,p_1) \diamond p_1.p_2.\mathsf{Buffer}$$

$$T_{10} \equiv \mathsf{buffer}(p_2,d,p_1) \diamond m_1,\ m_2 \diamond p_1.p_2.\mathsf{Buffer} \longrightarrow \\ m_1,\ m_2 \diamond \mathsf{output}(p_2,d,p_1) \diamond p_1.p_2.\mathsf{Buffer}$$

$$T_{11} \equiv \mathsf{connect}(p_2,p_1) \diamond m_1,\ \mathsf{open}(p_2) \diamond m_2 \longrightarrow \\ m_1,\ \mathsf{open}(p_2) \diamond \mathsf{join} \diamond m_2 \parallel (\mathsf{serve}(p_1))^\star \diamond p_2$$

$$T_{12} \equiv \mathsf{disconnect}(p_2,p_1) \diamond m_1,\ (\mathsf{serve}(p_1))^\star \diamond p_2,\ \mathsf{open}(p_2) \diamond \mathsf{join} \diamond m_2 \longrightarrow \\ m_1,\ \mathsf{open}(p_2) \diamond m_2$$

$$T_{13} \equiv \mathsf{request}(p_2,\{d_1,d_2\},p_1) \diamond m,\ (\mathsf{serve}(p_1))^\star \diamond p_2 \longrightarrow \\ \mathsf{output}(p_2,d_1,p_1) \diamond \mathsf{input}(p_2,d_2,p_1) \diamond m,\ \mathsf{serve}(p_1) \diamond (\mathsf{serve}(p_1))^\star \diamond p_2$$

$$T_{14} \equiv \mathsf{serve}(p) \diamond m \diamond \mathsf{DA} \longrightarrow \mathsf{input}(p,\mathit{identifier},\mathsf{DA}) \diamond \mathsf{output}(p,\mathit{doctrine},\mathsf{DA}) \diamond m \diamond \mathsf{DA}$$

$$T_{15} \equiv \mathsf{serve}(p) \diamond m \diamond \mathsf{TS} \longrightarrow \mathsf{input}(p,\mathit{identifier},\mathsf{TS}) \diamond \mathsf{output}(p,\mathit{track},\mathsf{TS}) \diamond m \diamond \mathsf{TS}$$

$$T_{16} \equiv \mathsf{serve}(p) \diamond m \diamond \mathsf{DV} \longrightarrow \\ \mathsf{input}(p,\{\mathit{identifier}_1,\mathit{identifier}_2\},\mathsf{DV}) \\ \diamond\ (\mathsf{request}(\mathsf{DA},\{\mathit{identifier}_1,\mathit{doctrine}\},\mathsf{DV}) + \mathsf{request}(\mathsf{TS},\{\mathit{identifier}_2,\mathit{track}\},\mathsf{DV})) \\ \diamond\ \mathsf{output}(p,\mathit{map},\mathsf{DV}) \diamond m \diamond \mathsf{DV}$$

$$T_{17} \equiv \mathsf{GS},\ \mathsf{GS.DR.Buffer},\ (m_1)^\star \diamond m_2 \diamond \mathsf{DR},\ (m_3)^\star \diamond m_4 \diamond \mathsf{DS} \longrightarrow \\ \mathsf{GS},\ \mathsf{GS.DR.Buffer},\ m_2 \diamond \mathsf{DR},\ m_4 \diamond \mathsf{DS}$$

$$T_{18} \equiv \mathsf{DS},\ \mathsf{open}(\mathsf{DV}) \diamond \mathsf{closed} \diamond m \diamond \mathsf{DV} \longrightarrow \mathsf{DS},\ m \diamond \mathsf{DV}$$

$$T_{19} \equiv \mathsf{DV},\ \mathsf{open}(\mathsf{DA}) \diamond \mathsf{closed} \diamond \mathsf{DA},\ \mathsf{open}(\mathsf{TS}) \diamond \mathsf{closed} \diamond \mathsf{TS} \longrightarrow \mathsf{DV},\ \mathsf{DA},\ \mathsf{TS}$$

B.2 Second Variant: Bundles

$$M ::= P \mid C \mid M \diamond M \mid M^\star \mid M \parallel M$$

$$P ::= \mathsf{EC} \mid \mathsf{DV} \mid \mathsf{DS} \mid \mathsf{DA} \mid \mathsf{TS} \mid \mathsf{GS} \mid \mathsf{DR}$$

$$\begin{aligned} D &::= \mathit{intersection} \mid \mathit{track} \mid \mathit{action} \mid \mathit{map} \mid \mathit{rule} \mid \\ &\quad\; \mathit{command} \mid \mathit{doctrine} \mid \mathsf{signal} \mid \{D'\} \\ D' &::= D \mid D, D' \end{aligned}$$

$$\begin{aligned} C ::= {} & \mathsf{input}(P,D,P) \mid \mathsf{output}(P,D,P) \mid \mathsf{connect}(P,P) \mid \mathsf{disconnect}(P,P) \mid \\ & \mathsf{open}(P) \mid \mathsf{join} \mid \mathsf{closed} \mid \mathsf{request}(P,D,P) \mid \mathsf{serve}(P) \mid \\ & \mathsf{buffer}(P,D,P) \mid P.P.\mathsf{Buffer} \mid C + C \end{aligned}$$

$$\begin{aligned} S_1 = {} & (\mathsf{output}(\mathsf{DA},\{\mathit{rule},\ldots\},\mathsf{EC}) + \mathsf{output}(\mathsf{DV},\{\mathit{command},\ldots\},\mathsf{EC}) \\ & \quad + \mathsf{output}(\mathsf{TS},\{\mathit{track},\ldots\},\mathsf{EC})) \diamond \mathsf{EC}, \\ & \mathsf{input}(\mathsf{EC},\{\mathit{rule},\ldots\},\mathsf{DA}) \diamond \mathsf{open}(\mathsf{DA}) \diamond \mathsf{closed} \diamond \mathsf{DA}, \\ & \mathsf{input}(\mathsf{EC},\{\mathit{track},\ldots\},\mathsf{TS}) \diamond \mathsf{open}(\mathsf{TS}) \diamond \mathsf{closed} \diamond \mathsf{TS}, \\ & \mathsf{input}(\mathsf{EC},\{\mathit{command},\ldots\},\mathsf{DV}) \diamond \mathsf{open}(\mathsf{DV}) \diamond \mathsf{closed} \diamond \mathsf{DV}, \\ & (\mathsf{connect}(\mathsf{DA},\mathsf{GS}) + \mathsf{connect}(\mathsf{TS},\mathsf{GS})) \\ & \quad \diamond ((\mathsf{request}(\mathsf{DA},\mathit{doctrine},\mathsf{GS}) + \mathsf{request}(\mathsf{TS},\mathit{track},\mathsf{GS})) \\ & \quad \diamond \mathsf{buffer}(\mathsf{DR},\{\mathit{doctrine},\mathit{track},\mathit{intersection}\},\mathsf{GS}))^\star \\ & \quad \diamond (\mathsf{disconnect}(\mathsf{DA},\mathsf{GS}) + \mathsf{disconnect}(\mathsf{TS},\mathsf{GS})) \diamond \mathsf{GS}, \\ & (\mathsf{input}(\mathsf{GS},\{\mathit{doctrine},\mathit{track},\mathit{intersection}\},\mathsf{DR}) \diamond \mathsf{output}(\mathsf{DS},\{\mathit{doctrine},\mathit{track},\mathit{action}\},\mathsf{DR}))^\star \diamond \mathsf{DR}, \\ & \mathsf{connect}(\mathsf{DV},\mathsf{DS}) \\ & \quad \diamond (\mathsf{input}(\mathsf{DR},\{\mathit{doctrine},\mathit{track},\mathit{action}\},\mathsf{DS}) \diamond \mathsf{request}(\mathsf{DV},\{\{\mathit{doctrine},\mathit{track}\},\mathit{map}\},\mathsf{DS}))^\star \\ & \quad \diamond \mathsf{disconnect}(\mathsf{DV},\mathsf{DS}) \diamond \mathsf{DS}, \\ & \mathsf{GS.DR.Buffer} \end{aligned}$$

$$
\begin{aligned}
T_1 &\equiv m_1 \parallel m_2 \longrightarrow m_1,\, m_2\\
T_2 &\equiv (c_1 + c_2) \diamond m \longrightarrow c_1 \diamond c_2 \diamond m\\
T_3 &\equiv (c_1 + c_2) \diamond m \longrightarrow c_2 \diamond c_1 \diamond m\\
T_4 &\equiv (m_1)^\star \diamond m_2 \diamond \mathsf{GS} \longrightarrow m_1 \diamond (m_1)^\star \diamond m_2 \diamond \mathsf{GS}\\
T_5 &\equiv (m_1)^\star \diamond m_2 \diamond \mathsf{GS} \longrightarrow m_2 \diamond \mathsf{GS}\\
T_6 &\equiv (m_1)^\star \diamond \mathsf{DR},\, m_2 \diamond \mathsf{GS.DR.Buffer} \longrightarrow\\
&\qquad m_1 \diamond (m_1)^\star \diamond \mathsf{DR},\, m_2 \diamond \mathsf{GS.DR.Buffer}\\
T_7 &\equiv (m_1)^\star \diamond m_2 \diamond \mathsf{DS},\, \mathsf{output}(\mathsf{DS},d,\mathsf{DR}) \diamond m_3 \longrightarrow\\
&\qquad m_1 \diamond (m_1)^\star \diamond m_2 \diamond \mathsf{DS},\, \mathsf{output}(\mathsf{DS},d,\mathsf{DR}) \diamond m_3\\
T_8 &\equiv \mathsf{input}(p_2,d,p_1) \diamond m_1,\, \mathsf{output}(p_1,d,p_2) \diamond m_2 \longrightarrow m_1,\, m_2\\
T_9 &\equiv \mathsf{buffer}(p_2,d,p_1) \diamond m_1,\, p_1.p_2.\mathsf{Buffer} \longrightarrow\\
&\qquad m_1,\, \mathsf{output}(p_2,d,p_1) \diamond p_1.p_2.\mathsf{Buffer}\\
T_{10} &\equiv \mathsf{buffer}(p_2,d,p_1) \diamond m_1,\, m_2 \diamond p_1.p_2.\mathsf{Buffer} \longrightarrow\\
&\qquad m_1,\, m_2 \diamond \mathsf{output}(p_2,d,p_1) \diamond p_1.p_2.\mathsf{Buffer}\\
T_{11} &\equiv \mathsf{connect}(p_2,p_1) \diamond m_1,\, \mathsf{open}(p_2) \diamond m_2 \longrightarrow\\
&\qquad m_1,\, \mathsf{open}(p_2) \diamond \mathsf{join} \diamond m_2 \parallel (\mathsf{serve}(p_1))^\star \diamond p_2\\
T_{12} &\equiv \mathsf{disconnect}(p_2,p_1) \diamond m_1,\, (\mathsf{serve}(p_1))^\star \diamond p_2,\, \mathsf{open}(p_2) \diamond \mathsf{join} \diamond m_2 \longrightarrow\\
&\qquad m_1,\, \mathsf{open}(p_2) \diamond m_2\\
T_{13} &\equiv \mathsf{request}(p_2,d,p_1) \diamond m,\, (\mathsf{serve}(p_1))^\star \diamond p_2 \longrightarrow\\
&\qquad \mathsf{output}(p_2,\mathsf{signal},p_1) \diamond \mathsf{input}(p_2,d,p_1) \diamond m,\, \mathsf{serve}(p_1) \diamond (\mathsf{serve}(p_1))^\star \diamond p_2\\
T'_{13} &\equiv \mathsf{request}(p_2,\{d_1,d_2\},p_1) \diamond m,\, (\mathsf{serve}(p_1))^\star \diamond p_2 \longrightarrow\\
&\qquad \mathsf{output}(p_2,d_1,p_1) \diamond \mathsf{input}(p_2,d_2,p_1) \diamond m,\, \mathsf{serve}(p_1) \diamond (\mathsf{serve}(p_1))^\star \diamond p_2\\
T_{14} &\equiv \mathsf{serve}(p) \diamond m \diamond \mathsf{DA} \longrightarrow \mathsf{input}(p,\mathsf{signal},\mathsf{DA}) \diamond \mathsf{output}(p,\mathit{doctrine},\mathsf{DA}) \diamond m \diamond \mathsf{DA}\\
T_{15} &\equiv \mathsf{serve}(p) \diamond m \diamond \mathsf{TS} \longrightarrow \mathsf{input}(p,\mathsf{signal},\mathsf{TS}) \diamond \mathsf{output}(p,\mathit{track},\mathsf{TS}) \diamond m \diamond \mathsf{TS}\\
T_{16} &\equiv \mathsf{serve}(p) \diamond m \diamond \mathsf{DV} \longrightarrow\\
&\qquad \mathsf{input}(p,\{\mathit{doctrine},\mathit{track}\},\mathsf{DV}) \diamond \mathsf{output}(p,\mathit{map},\mathsf{DV}) \diamond m \diamond \mathsf{DV}\\
T_{17} &\equiv \mathsf{GS},\, \mathsf{GS.DR.Buffer},\, (m_1)^\star \diamond \mathsf{DR},\, (m_2)^\star \diamond m_3 \diamond \mathsf{DS} \longrightarrow\\
&\qquad \mathsf{GS},\, \mathsf{GS.DR.Buffer},\, \mathsf{DR},\, m_3 \diamond \mathsf{DS}\\
T_{18} &\equiv \mathsf{DS},\, \mathsf{open}(\mathsf{DV}) \diamond \mathsf{closed} \diamond \mathsf{DV} \longrightarrow \mathsf{DS},\, \mathsf{DV}\\
T_{19} &\equiv \mathsf{DV},\, \mathsf{open}(\mathsf{DA}) \diamond \mathsf{closed} \diamond \mathsf{DA},\, \mathsf{open}(\mathsf{TS}) \diamond \mathsf{closed} \diamond \mathsf{TS} \longrightarrow \mathsf{DV},\, \mathsf{DA},\, \mathsf{TS}
\end{aligned}
$$

B.3 Third Variant: Lockstep

$$M ::= P \mid C \mid M \diamond M \mid M^\star \mid M \parallel M$$

$$P ::= \mathsf{EC} \mid \mathsf{DV} \mid \mathsf{DS} \mid \mathsf{DA} \mid \mathsf{TS} \mid \mathsf{GS} \mid \mathsf{DR}$$

$$\begin{aligned} D &::= \mathit{intersection} \mid \mathit{track} \mid \mathit{action} \mid \mathit{map} \mid \mathit{rule} \mid \\ &\quad\ \mathit{command} \mid \mathit{doctrine} \mid \mathsf{signal} \mid \{D'\} \\ D' &::= D \mid D, D' \end{aligned}$$

$$\begin{aligned} C ::= {} & \mathsf{input}(P,D,P) \mid \mathsf{output}(P,D,P) \mid \mathsf{connect}(P,P) \mid \mathsf{disconnect}(P,P) \mid \\ & \mathsf{open}(P) \mid \mathsf{join} \mid \mathsf{closed} \mid \mathsf{request}(P,D,P) \mid \mathsf{serve}(P) \mid \\ & \mathsf{buffer}(P,D,P) \mid P.P.\mathsf{Buffer} \mid C + C \\ & \mathsf{token}(P,D) \mid \mathsf{spent}(P,D) \mid \mathsf{step} \mid \mathsf{current}(D) \end{aligned}$$

$$\begin{aligned} S_1 = {} & (\mathsf{output}(\mathsf{DA},\{\textsf{\textit{rule}},\ldots\},\mathsf{EC}) + \mathsf{output}(\mathsf{DV},\{\textsf{\textit{command}},\ldots\},\mathsf{EC}) \\ & \quad + \mathsf{output}(\mathsf{TS},\{\textsf{\textit{track}},\ldots\},\mathsf{EC})) \diamond \mathsf{EC}, \\ & \mathsf{input}(\mathsf{EC},\{\textsf{\textit{rule}},\ldots\},\mathsf{DA}) \diamond (\mathsf{open}(\mathsf{DA}) \diamond \mathsf{closed} \diamond \mathsf{DA} \parallel \mathsf{step} \diamond \mathsf{DA}), \\ & \mathsf{input}(\mathsf{EC},\{\textsf{\textit{track}},\ldots\},\mathsf{TS}) \diamond (\mathsf{open}(\mathsf{TS}) \diamond \mathsf{closed} \diamond \mathsf{TS} \parallel \mathsf{step} \diamond \mathsf{TS}), \\ & \mathsf{input}(\mathsf{EC},\{\textsf{\textit{command}},\ldots\},\mathsf{DV}) \diamond (\mathsf{connect}(\mathsf{DA},\mathsf{DV}) + \mathsf{connect}(\mathsf{TS},\mathsf{DV})) \\ & \quad \diamond \mathsf{open}(\mathsf{DV}) \diamond \mathsf{closed} \diamond (\mathsf{disconnect}(\mathsf{DA},\mathsf{DV}) + \mathsf{disconnect}(\mathsf{TS},\mathsf{DV})) \diamond \mathsf{DV}, \\ & (\mathsf{connect}(\mathsf{DA},\mathsf{GS}) + \mathsf{connect}(\mathsf{TS},\mathsf{GS})) \\ & \quad \diamond ((\mathsf{request}(\mathsf{DA},\textsf{\textit{doctrine}},\mathsf{GS}) + \mathsf{request}(\mathsf{TS},\textsf{\textit{track}},\mathsf{GS})) \diamond \mathsf{buffer}(\mathsf{DR},\textsf{\textit{intersection}},\mathsf{GS}))^\star \\ & \quad \diamond (\mathsf{disconnect}(\mathsf{DA},\mathsf{GS}) + \mathsf{disconnect}(\mathsf{TS},\mathsf{GS})) \diamond \mathsf{GS}, \\ & (\mathsf{connect}(\mathsf{DA},\mathsf{DR}) + \mathsf{connect}(\mathsf{TS},\mathsf{DR})) \\ & \quad \diamond (\mathsf{input}(\mathsf{GS},\textsf{\textit{intersection}},\mathsf{DR}) \diamond (\mathsf{request}(\mathsf{DA},\textsf{\textit{doctrine}},\mathsf{DR}) + \mathsf{request}(\mathsf{TS},\textsf{\textit{track}},\mathsf{DR})) \\ & \qquad \diamond \mathsf{output}(\mathsf{DS},\textsf{\textit{action}},\mathsf{DR}))^\star \\ & \quad \diamond (\mathsf{disconnect}(\mathsf{DA},\mathsf{DR}) + \mathsf{disconnect}(\mathsf{TS},\mathsf{DR})) \diamond \mathsf{DR}, \\ & (\mathsf{connect}(\mathsf{DA},\mathsf{DS}) + \mathsf{connect}(\mathsf{TS},\mathsf{DS}) + \mathsf{connect}(\mathsf{DV},\mathsf{DS})) \\ & \quad \diamond (\mathsf{input}(\mathsf{DR},\textsf{\textit{action}},\mathsf{DS}) \\ & \quad \diamond (\mathsf{request}(\mathsf{DV},\textsf{\textit{map}},\mathsf{DS}) + \mathsf{request}(\mathsf{DA},\textsf{\textit{doctrine}},\mathsf{DS}) + \mathsf{request}(\mathsf{TS},\textsf{\textit{track}},\mathsf{DS})))^\star \\ & \quad \diamond (\mathsf{disconnect}(\mathsf{DA},\mathsf{DS}) + \mathsf{disconnect}(\mathsf{TS},\mathsf{DS}) + \mathsf{disconnect}(\mathsf{DV},\mathsf{DS})) \diamond \mathsf{DS}, \\ & \mathsf{token}(\mathsf{DA},\mathsf{GS}),\ \mathsf{token}(\mathsf{DA},\mathsf{DR}),\ \mathsf{token}(\mathsf{DA},\mathsf{DV}),\ \mathsf{token}(\mathsf{DA},\mathsf{DS}), \\ & \mathsf{token}(\mathsf{TS},\mathsf{GS}),\ \mathsf{token}(\mathsf{TS},\mathsf{DR}),\ \mathsf{token}(\mathsf{TS},\mathsf{DV}),\ \mathsf{token}(\mathsf{TS},\mathsf{DS}), \\ & \mathsf{GS.DR.Buffer} \end{aligned}$$

$$
\begin{aligned}
T_1 &\equiv m_1 \parallel m_2 \longrightarrow m_1,\, m_2 \\
T_2 &\equiv (c_1 + c_2) \diamond m \longrightarrow c_1 \diamond c_2 \diamond m \\
T_3 &\equiv (c_1 + c_2) \diamond m \longrightarrow c_2 \diamond c_1 \diamond m \\
T_4 &\equiv (m_1)^\star \diamond m_2 \diamond \mathsf{GS} \longrightarrow m_1 \diamond (m_1)^\star \diamond m_2 \diamond \mathsf{GS} \\
T_5 &\equiv (m_1)^\star \diamond m_2 \diamond \mathsf{GS} \longrightarrow m_2 \diamond \mathsf{GS} \\
T_6 &\equiv (m_1)^\star \diamond m_2 \diamond \mathsf{DR},\, m_3 \diamond \mathsf{GS.DR.Buffer} \longrightarrow \\
&\qquad m_1 \diamond (m_1)^\star \diamond m_2 \diamond \mathsf{DR},\, m_3 \diamond \mathsf{GS.DR.Buffer} \\
T_7 &\equiv (m_1)^\star \diamond m_2 \diamond \mathsf{DS},\, \mathsf{output}(\mathsf{DS},d,\mathsf{DR}) \diamond m_3 \longrightarrow \\
&\qquad m_1 \diamond (m_1)^\star \diamond m_2 \diamond \mathsf{DS},\, \mathsf{output}(\mathsf{DS},d,\mathsf{DR}) \diamond m_3 \\
T_8 &\equiv \mathsf{input}(p_2,d,p_1) \diamond m_1,\, \mathsf{output}(p_1,d,p_2) \diamond m_2 \longrightarrow m_1,\, m_2 \\
T_9 &\equiv \mathsf{buffer}(p_2,d,p_1) \diamond m_1,\, p_1.p_2.\mathsf{Buffer} \longrightarrow \\
&\qquad m_1,\, \mathsf{output}(p_2,d,p_1) \diamond p_1.p_2.\mathsf{Buffer} \\
T_{10} &\equiv \mathsf{buffer}(p_2,d,p_1) \diamond m_1,\, m_2 \diamond p_1.p_2.\mathsf{Buffer} \longrightarrow \\
&\qquad m_1,\, m_2 \diamond \mathsf{output}(p_2,d,p_1) \diamond p_1.p_2.\mathsf{Buffer} \\
T_{11} &\equiv \mathsf{connect}(p_2,p_1) \diamond m_1,\, \mathsf{open}(p_2) \diamond m_2 \longrightarrow \\
&\qquad m_1,\, \mathsf{open}(p_2) \diamond \mathsf{join} \diamond m_2 \parallel (\mathsf{serve}(p_1))^\star \diamond p_2 \\
T_{12} &\equiv \mathsf{disconnect}(p_2,p_1) \diamond m_1,\, (\mathsf{serve}(p_1))^\star \diamond p_2,\, \mathsf{open}(p_2) \diamond \mathsf{join} \diamond m_2 \longrightarrow \\
&\qquad m_1,\, \mathsf{open}(p_2) \diamond m_2 \\
T_{13} &\equiv \mathsf{request}(p_2,d,p_1) \diamond m,\, (\mathsf{serve}(p_1))^\star \diamond p_2 \longrightarrow \\
&\qquad \mathsf{output}(p_2,\mathsf{signal},p_1) \diamond \mathsf{input}(p_2,d,p_1) \diamond m,\, \mathsf{serve}(p_1) \diamond (\mathsf{serve}(p_1))^\star \diamond p_2 \\
T_{14} &\equiv \mathsf{serve}(p) \diamond m \diamond \mathsf{DA},\, \mathsf{current}(\textit{doctrine}) \diamond \mathsf{DA},\, \mathsf{token}(\mathsf{DA},p) \longrightarrow \\
&\qquad \mathsf{input}(p,\mathsf{signal},\mathsf{DA}) \diamond \mathsf{output}(p,\textit{doctrine},\mathsf{DA}) \diamond m \diamond \mathsf{DA}, \\
&\qquad \mathsf{current}(\textit{doctrine}) \diamond \mathsf{DA},\, \mathsf{spent}(\mathsf{DA},p) \\
T_{15} &\equiv \mathsf{serve}(p) \diamond m \diamond \mathsf{TS},\, \mathsf{current}(\textit{track}) \diamond \mathsf{TS},\, \mathsf{token}(\mathsf{TS},p) \longrightarrow \\
&\qquad \mathsf{input}(p,\mathsf{signal},\mathsf{TS}) \diamond \mathsf{output}(p,\textit{track},\mathsf{TS}) \diamond m \diamond \mathsf{TS}, \\
&\qquad \mathsf{current}(\textit{track}) \diamond \mathsf{TS},\, \mathsf{spent}(\mathsf{TS},p) \\
T_{16} &\equiv \mathsf{serve}(p) \diamond m \diamond \mathsf{DV} \longrightarrow \\
&\qquad \mathsf{input}(p,\mathsf{signal},\mathsf{DV}) \\
&\qquad \diamond\, (\mathsf{request}(\mathsf{DA},\textit{doctrine},\mathsf{DV}) + \mathsf{request}(\mathsf{TS},\textit{track},\mathsf{DV})) \\
&\qquad \diamond\, \mathsf{output}(p,\textit{map},\mathsf{DV}) \diamond m \diamond \mathsf{DV} \\
T_{17} &\equiv \mathsf{GS},\, \mathsf{GS.DR.Buffer},\, (m_1)^\star \diamond m_2 \diamond \mathsf{DR},\, (m_3)^\star \diamond m_4 \diamond \mathsf{DS} \longrightarrow \\
&\qquad \mathsf{GS},\, \mathsf{GS.DR.Buffer},\, m_2 \diamond \mathsf{DR},\, m_4 \diamond \mathsf{DS} \\
T_{18} &\equiv \mathsf{DS},\, \mathsf{open}(\mathsf{DV}) \diamond \mathsf{closed} \diamond m \diamond \mathsf{DV} \longrightarrow \mathsf{DS},\, m \diamond \mathsf{DV} \\
T_{19} &\equiv \mathsf{DV},\, \mathsf{open}(\mathsf{DA}) \diamond \mathsf{closed} \diamond \mathsf{DA},\, \mathsf{open}(\mathsf{TS}) \diamond \mathsf{closed} \diamond \mathsf{TS} \longrightarrow \mathsf{DV},\, \mathsf{DA},\, \mathsf{TS} \\
T_{20} &\equiv \mathsf{step} \diamond \mathsf{DA},\, \mathsf{step} \diamond \mathsf{TS} \longrightarrow \mathsf{current}(\textit{doctrine}) \diamond \mathsf{DA},\, \mathsf{current}(\textit{track}) \diamond \mathsf{TS} \\
T_{21} &\equiv \mathsf{spent}(p,\mathsf{GS}),\, \mathsf{spent}(p,\mathsf{DR}),\, \mathsf{spent}(p,\mathsf{DV}),\, \mathsf{spent}(p,\mathsf{DS}),\, \mathsf{current}(d) \diamond p \longrightarrow \\
&\qquad \mathsf{token}(p,\mathsf{GS}),\, \mathsf{token}(p,\mathsf{DR}),\, \mathsf{token}(p,\mathsf{DV}),\, \mathsf{token}(p,\mathsf{DS}),\, \mathsf{step} \diamond p
\end{aligned}
$$