A Unifying Approach to the Security of Distributed and Multi-Threaded Programs*

Heiko Mantel†
German Research Center for Artificial Intelligence (DFKI)
Stuhlsatzenhausweg 3
66123 Saarbrücken, Germany
mantel@dfki.de

Andrei Sabelfeld‡
Department of Computer Science
Upson Hall, Cornell University
Ithaca, NY 14853, USA
andrei@cs.cornell.edu

Abstract

The security of computation at the level of a specific programming language and the security of complex systems at a more abstract level are two major areas of current security research. With the objective to integrate the two, this article proposes an adequate translation of a timing-sensitive security property for simple multi-threaded programs into a more general security framework. *Soundness* and *completeness* of the translation guarantee that the trace-based specification of the translation of a multi-threaded program is secure if and only if the original program is secure. Finally, the translation is extended to a distributed setting, and it is demonstrated how to derive global security of the overall system from local security of each thread. The translation is presented as a two-step process where the first step is independent from the concrete programming language.

1 Introduction

1.1 Motivation

An important step in the specification of secure information flow in a complex distributed system where local parts are written in a particular programming language is to combine two types of security. Namely, the first type is the security of communication between local computations and the second type is the security of the local

*To appear in *Journal of Computer Security*, 2002. © IOS Press 2002. This is an extended version of the article [27] that appeared in *Proceedings of the 14th IEEE Computer Security Foundations Workshop*, Cape Breton, Nova Scotia, Canada, June 11–13, 2001.

†This work was partly supported by the German Research Foundation (DFG).

‡This research was partly supported by the Department of the Navy, Office of Naval Research, ONR Grant N00014-01-1-0968. Any opinions, findings, conclusions, or recommendations contained in this material are those of the authors and do not necessarily reflect the views of the Office of Naval Research. This work was partly supported by TFR while this author was with the Department of Computer Science, Chalmers University of Technology and University of Göteborg, Sweden.

computations themselves. The former is often defined as security of an event-based system (as in the underlying model of [22]) whereas the latter relies on the security specification of the programming language (as in the underlying model of [38] for a simple imperative multi-threaded language). Embracing the two kinds of security into a single security framework is the motivation of this article.

1.2 Background

There is a large body of research on information flow control aiming at specifying, verifying, and analyzing security. In the traditional abstract view, security is often defined for an abstract *trace-based* model of computation. In particular, a system can be represented as a set of its traces and, thus, security is a property that can be true or false for a given set of traces. In a distributed setting, these traces can be viewed as sequences of events like, e.g., communication of local processes in a distributed network.

Many different approaches to this type of general information flow control have been proposed. Nondeducibility was developed by Sutherland [43] because the original definition of noninterference [15] was based on a system model, deterministic state machines, that is not adequate for modeling nondeterministic systems. Motivated by the observation that nondeducibility is not preserved under composition, McCullough suggested restrictiveness, a composable security property [28]. Subsequently, numerous other possibilistic information flow properties have been proposed (e.g., [14, 20, 18, 46, 33, 34]). To date, it appears that none of these information flow properties is optimal for all purposes. Rather, it depends on the particular application, which of the various information flow properties is most appropriate. The desire to understand the existing information flow properties and their metaproperties better has led to detailed comparisons and uniform frameworks [31, 12, 49, 22].

Another line of research that is becoming increasingly popular is information flow control in a setting of a concrete programming language (see [37] for a state-of-the-art survey). The efforts in this area are focused on determining whether a given program written in a particular programming language has secure information flow. More concrete assumptions are usually made about local computations. For example, one might assume that the program runs on a partition of data on high (private) and low (public) security data (although a more general lattice of security levels can be considered). The program is not trusted (possibly received over the Internet). The program's low output is publicly available (e.g., sent over the Internet) as well as, perhaps, timing information about the program's execution (e.g., times when the program makes Internet accesses are observable).

Originating from early work of Denning [10, 11] and Cohen [7, 8], secure information flow in programming languages received its recent reincarnation in work of Volpano et al. [45] with the main contribution being soundness proofs for a Denning-style security analysis. Many other researchers have investigated the problem of secure information flow. This work includes Joshi and Leino's equational specification [21], a single calculus for security, binding-time analysis, program slicing and call-tracking (DCC) by Abadi et al. [1], Heintze and Riecke's Secure Lambda Calculus (Slam) [19], Volpano and Smith's investigations on security of concurrent programs

[42, 44], and Sabelfeld and Sands's compositional security specifications for sequential [39] and multi-threaded programs [38].

The security formalization in the studies mentioned is founded on the extensional approach to security, namely *noninterference* [15]. The idea behind noninterference is that a system is considered secure if high inputs do not interfere with low-observable behavior of the system (low outputs, timing, etc.). It has often been claimed that extensional programming-language-based security can be viewed as a form of noninterference (e.g., in [45]), especially since the revival of the interest in language-based security. Nevertheless, for the language-based extensional security models that have been proposed since the mid-nineties a rigorous connection to noninterference-like properties has not so far been established to the best of our knowledge.[1] This article is a step in this direction.

1.3 Foreground

Our choice for the abstract event-based framework is *MAKS*, the *modular assembly kit for security properties* [22]. Guided by *MAKS*, we can pick an appropriate security property from the assembly kit rather than inventing a new one. This also allows for combining the security of programs with the security of other components in a (potentially distributed) system using *MAKS* as an interface. This means integrating programming-language-based security at a higher level of abstraction, opening the opportunity for plugging the security of subsystems written in a particular programming language to the global security of the system defined in a general event-based framework. Finally, *MAKS* enjoys a number of useful extensions including local verification conditions [23], intransitive security policies [24], refinement operators [25], and compositionality results [26], which potentially enables us to use these verification techniques, to apply intransitive security policies, and to do stepwise development in the setting of secure information flow in multi-threaded programs (although, besides compositionality, these issues are outside the scope of the present article).

The focus of this article is on a simple multi-threaded while-language (MWL) and a timing-sensitive security specification (*strong security* [38]) that implies robust security independently of a particular scheduler. We translate MWL programs into state-event systems, pick an appropriate definition of security from the assembly kit, and establish a precise correspondence between the security of MWL programs and their translations. Namely, that the translation is *sound* in the sense that the translation of any secure MWL program is secure as a state-event system; and *complete* in the sense that if the translation of an MWL program is secure as a state-event system then the original program is secure.

Benefits of such a rigorous connection between the two types of security become evident when extending MWL programming to a truly distributed setting. Given a distributed collection of programs (each equipped with its own memory) that interact by message passing, we have a complex global system with the ultimate security requirement being *global security*. The underlying semantic model for a global distributed computation is event-based. Accordingly, the global security condition is expressed

[1]Not forgetting McLean's article [30] from the early nineties, that used functional specifications as an intermediate step when proving noninterference for programs in a sequential imperative language.

as a security property from the assembly kit. However, showing such a property is greatly facilitated by the rigorous relation established in this article. Due to such a relation, global security can be *derived* from *local security* of all programs in the system. In particular, we will show that in the case of distributed MWL, global security follows from the local security of each individual thread, as a basic part of the overall system. As a direct consequence, language-based techniques (e.g., security-type systems [45, 19, 42, 1, 44, 2, 38, 39, 35] or security verification [21]) applied to individual threads, can be used to guarantee the global security of the overall distributed system.

1.4 Overview

After recalling some preliminaries in Section 2, we introduce the concept of thread pools in Section 3. In Section 4, we specialize this generic model according to the syntax and semantics of the MWL programming language. That this specialization indeed reflects the semantics of MWL, is ensured by a collection of theorems in Section 5. The key contribution of our translation is that it preserves the specification of secure information flow. Section 6 shows that a thread pool is considered to be secure in the MWL programming language if and only if the corresponding state-event system is also considered to be secure in the assembly kit. In Section 7, we enrich MWL with message-passing primitives and adapt the security condition in order to support distributed programming. The thread pool model is extended accordingly and we show that the security condition for the distributed extension of MWL coincides with a trace-based security property that is preserved under the composition of thread pools. We conclude by a discussion in Section 8.

2 Preliminaries

2.1 System Specifications

The behavior of systems can often be adequately specified by the set of its possible execution sequences. We follow this trace-based approach throughout this article (with the exception of parts where we use a concrete programming language). A *trace* is a sequence of events that models a possible execution sequence of the system. An *event* is an atomic action like, e.g., the sending or receiving of a message on some channel. We distinguish between input, internal, and output events. The underlying intuition is that input events are controlled by the environment of a system while internal and output events are controlled by the system. Input and output events constitute the interface of a system. The distinction between input and output events is somewhat fuzzy. When a system is capable of preventing the occurrence of input events, then this can be interpreted as a signal to the environment. To avoid this kind of communication, *input totality* is often assumed, i.e., that a system cannot prevent the occurrence of input events. However, a limitation to input total systems appears to be quite restrictive. Therefore, we refrain from making the assumption of input totality in this article.

For specifying systems, we do not define the set of traces directly but rather use states as an auxiliary concept. This allows us to define the possible traces inductively by a transition relation. The system model, we use for specification, are state-event systems [23]. This system model allows for the specification of nondeterministic systems where the nondeterminism is reflected by the choice between different events that are enabled. For simplicity, any nondeterminism in the effects of events is ruled out.

Definition 1 *Let S be a set of states, E be a set of events, and $T \subseteq S \times E \times S$ be a transition relation. A* state-event system *SES is a tuple (S, s_0, E, I, O, T) where $s_0 \in S$ is the initial state and $I, O \subseteq E$, respectively, are the input and output events. Moreover, it is assumed that $I \cap O = \emptyset$ holds and that for a given state s and event e there is at most one state s' with $(s, e, s') \in T$.*[2]

Note that the set of internal events, i.e., $E \setminus (I \cup O)$, may be nonempty.

Example 1 *A random generator that outputs a sequence of random natural numbers and then terminates can be specified by the state-event system SES $= (S, s_0, E, I, O, T)$. SES is defined by $S = \{s_0, s_f\}$, $E = O \cup \{\mathit{term}\}$, $I = \emptyset$, $O = \{\mathit{out}(n) \mid n \in \mathbb{N}\}$, $T = \{(s_0, \mathit{out}(n), s_0) \mid n \in \mathbb{N}\} \cup \{(s_0, \mathit{term}, s_f)\}$. Possible traces for this state-event system include, e.g., $\langle \mathit{out}(42)\rangle$ and $\langle \mathit{out}(17).\mathit{out}(42).\mathit{term}\rangle$.*

Let $s_1, s_2, s' \in S$, $e \in E$, and $\gamma \in E^*$. Instead of $(s_1, e, s_2) \in T$ we sometimes use the notation $s_1 \xrightarrow{e}_T s_2$. For multi-event transitions, we use the notation $s_1 \overset{\gamma}{\Longrightarrow}_T s'$. If T is obvious from the context then we omit the index and write $s_1 \xrightarrow{e} s_2$ or $s_1 \overset{\gamma}{\Longrightarrow} s'$. The relation $\overset{\gamma}{\Longrightarrow}_T$ is formally defined as follows:

$$\begin{array}{ll} s_1 \overset{\langle\rangle}{\Longrightarrow}_T s' & \text{, if } s_1 = s' \\ s_1 \overset{\langle e\rangle.\gamma}{\Longrightarrow}_T s' & \text{, if } \exists s_2 \in S. s_1 \xrightarrow{e}_T s_2 \wedge s_2 \overset{\gamma}{\Longrightarrow}_T s' \end{array}$$

A sequence $\tau \in E^*$ of events is a *trace* of a state-event system *SES* $= (S, s_0, E, I, O, T)$ if it is accepted in the initial state, i.e., if $\exists s' \in S. s_0 \overset{\tau}{\Longrightarrow}_T s'$. The set of all traces that are possible for *SES* is denoted by $\mathit{Tr}_{\mathit{SES}}$. We omit the index and simply write *Tr* if the state-event system is obvious from the context. The tuple $\mathit{ES}_{\mathit{SES}} = (E, I, O, \mathit{Tr}_{\mathit{SES}})$ is referred to as the *event system* [20] corresponding to *SES*. A state $s \in S$ is *reachable* if there exists a sequence $\tau \in E^*$ such that $s_0 \overset{\tau}{\Longrightarrow} s$. The *projection* $\alpha|_{E'}$ of a sequence $\alpha \in E^*$ to the events in $E' \subseteq E$ results from α by deleting all events *not* in E'.

In complex systems, communication between components is done by synchronization on the occurrence of shared events (usually output events of the one component that are input events of others). We define the composition of state-event systems.

Definition 2 *Given an index set J and, for each $j \in J$, a state-event system $\mathit{SES}^j = (S^j, s_0^j, E^j, I^j, O^j, T^j)$ such that for all $k, l \in J$ with $k \neq l$ holds $E^k \cap E^l \subseteq (I^k \cap O^l) \cup (I^l \cap O^k)$. We define the* composition *of the state-event systems $||_{j \in J}\, \mathit{SES}^j$ to be*

[2] Note that our system model is possibilistic, i.e., it abstracts from probabilities of occurrences of particular events. This is a common assumption that, e.g., has also been made in [43, 14, 28, 18, 20, 31, 12, 49, 34, 22]. For information flow properties that are based on a probabilistic system model, we refer to [29, 17].

the state-event system (S, s_0, E, I, O, T) *where*

$$\begin{aligned}
S &= \times_{j\in J} S^j\\
s_0 &= (s_0^j)_{j\in J}\\
E &= \cup_{j\in J} E^j\\
I &= \cup_{j\in J} I^j \setminus \cup_{j\in J} O^j\\
O &= \cup_{j\in J} O^j \setminus \cup_{j\in J} I^j\\
T &= \{(s,e,s') \in S \times E \times S \mid \forall j \in J. [(e \in E^j \wedge (s|_j, e, s'|_j) \in T^j)\\
&\qquad\qquad\qquad\qquad\qquad\qquad \vee (e \notin E^j \wedge s|_j = s'|_j)]\}
\end{aligned}$$

The state of a composed state-event system is a tuple of component states. $s|_j$ denotes the jth element of a state s of a composed *SES*. Note that the definition above, indeed, guarantees that communication between components is only possible by synchronization on occurrences of shared events, i.e., events that are output events of one component and input events of another component. Such communication events are considered to be internal events for the composed system.

The following theorem justifies the definition of the composition on state-event systems by relating it to the usual definition of composition for event systems (cf., e.g., [20, 26]). Essentially, the theorem ensures that $ES_{||_{j\in J}\ SES^j} = ||_{j\in J}\ ES_{SES^j}$ holds.

Theorem 1 *For an index set* J *and a collection of state-event systems* $(SES^j)_{j\in J}$ *we have* $Tr_{||_{j\in J}\ SES^j} = \{\tau \in E^* \mid \forall j \in J. \tau|_{E^j} \in Tr_{SES^j}\}$.

2.2 Security Properties

Many security requirements can be expressed as restrictions on the information flow within a system. To express confidentiality or integrity by such restrictions is the key idea of information flow control.

The assembly kit *MAKS* that supports the uniform and modular representation of information flow properties has been previously proposed by one of the authors [22]. It simplifies the comparison among the existing security properties that are based on possibilistic information flow as well as a goal-directed construction of new ones. In *MAKS*, a *security property* consists of a set of views and a security predicate.[3]

A *view* [24] specifies restrictions on the permitted flow of information within a system. Formally, a view $\mathcal{V}$ (in a set of events E) is a triple (V, N, C) where $V, N, C \subseteq E$ are sets of events that form a disjoint partition of E.[4] Intuitively, a view describes the perspective of a (potentially malicious) observer of the system. For a given view $\mathcal{V} = (V, N, C)$, the set V specifies the events that are *visible* for the observer (or an attacker in the observer's guise). These events can be directly observed when they occur. Occurrences of all other events (i.e., events in $N \cup C$) are *not* directly observable.

[3]If desired, the set of views can be specified by a flow policy and a domain assignment using a graphical notation. This approach has been explained, e.g., in [24]. However, this possibility is not important for the purposes of the current article.

[4]I.e., $V \cap N = V \cap C = N \cap C = \emptyset$ and $V \cup N \cup C = E$.

Hence, if a trace $\tau \in \mathit{Tr}$ occurs then the observer only sees the projection $\tau|_V$. The set C specifies the set of events that are *confidential* for the observer. I.e., the observer must not be able to learn *any* information about occurrences of events in C (based on his observations and other knowledge he might have about the systems).[5] All other events (i.e., events in $V \cup N$) are *not* confidential. Since the sets V, N, C must form a disjoint partition of E, a view is completely determined by the sets V and C. The remaining events are collected in the set N. These events are *neither* visible *nor* confidential for the observer.

Example 2 *A view $\mathcal{V}_1 = (V, N, \emptyset)$ does not impose any restrictions on the information flow because there are no confidential events. A view $\mathcal{V}_2 = (\emptyset, N, C)$ specifies that nothing can be observed during system execution because the set of visible events is empty. Note that for all possible traces $\tau \in \mathit{Tr}$ holds $\tau|_\emptyset = \langle\rangle$. Clearly, $\mathcal{V}_1$ and $\mathcal{V}_2$ constitute two extreme cases.*

A more interesting case is specified by the view $\mathcal{V}_3 = (L, H\backslash HI, HI)$. Assuming a two-level security policy where each event is either classified as a high- or a low-level event, $\mathcal{V}_3$ states that low-level events (L) are visible and high-level input events (HI) are confidential. This view captures the original idea of noninterference, i.e., that high inputs do not interfere with low-observable behavior of a system.[6]

A *security predicate* [24] specifies under which conditions the requirements of a given view are satisfied for a set of traces.

Definition 3 *For a* view set *VS and a* security predicate *SP, a* security property (*VS*,*SP*) *is satisfied by an event system $ES = (E, I, O, \mathit{Tr})$ if $SP_{\mathcal{V}}(\mathit{Tr})$ holds for each view $\mathcal{V} \in VS$. A state-event system SES* satisfies *a security property if the corresponding event system ES_{SES} satisfies it.*

In *MAKS*, security predicates are composed by conjunction from one or more basic security predicates (abbreviated by *BSP*). For the purposes of the current article, only two *BSPs* are of interest: *backwards strict insertion of confidential events* (abbreviated by *BSI*) and *backwards strict deletion of confidential events* (abbreviated by *BSD*) [24].

For a view $\mathcal{V}$, *BSI* requires that the occurrence of an event from C does *not remove* possible V-observations. Consider the system after a trace β has occurred. Any V-observation $\overline{\alpha} \in V^*$ that is possible at this point must also be possible after $c \in C$ has occurred. Consequently, if the V-observation $\overline{\alpha}$ results from the sequence $\alpha \in (V \cup N)^*$, i.e., $\alpha|_V = \overline{\alpha}$, then some sequence $\alpha' \in (V \cup N)^*$ must be enabled after c has occurred where α' may differ from α only in events from N. Hence, $\alpha'|_V = \overline{\alpha} = \alpha|_V$ holds. For a given view $\mathcal{V} = (V, N, C)$, $BSI_{\mathcal{V}}(\mathit{Tr})$ is formally defined as follows:

$$
\begin{aligned}
BSI_{V,N,C}(\mathit{Tr}) \quad \equiv \quad & \forall \alpha, \beta \in E^*. \forall c \in C. \left((\beta.\alpha \in \mathit{Tr} \wedge \alpha|_C = \langle\rangle) \right. \\
& \left. \Longrightarrow \exists \alpha' \in E^*. (\alpha'|_V = \alpha|_V \wedge \alpha'|_C = \langle\rangle \wedge \beta.\langle c\rangle.\alpha' \in \mathit{Tr}) \right)
\end{aligned}
$$

[5] As usual in investigations of secure information flow, we assume that the observer has complete knowledge of the system specification. This is a worst-case assumption.

[6] The view $\mathcal{V}_3$ is appropriate for systems that operate on confidential data that they receive as input but do not generate new secrets internally (in the sense of ,e.g., a random-number generator).

For simplicity of reasoning about *BSI*, it requires only that a confidential event c can be inserted at a point that is not followed by any other confidential events ($\alpha|_C = \langle\rangle$) and that no other confidential events besides c are inserted ($\alpha'|_C = \langle\rangle$).

If $BSI_{\mathcal{V}}(Tr)$ does not hold then it might be possible for an adversary to infer from some V-observation $\overline{\alpha}$ that c cannot have occurred. The security guarantee provided by *BSI* is: if an adversary observes $\overline{\alpha}$ starting in some state then he or she cannot deduce that a confidential event c *has not* occurred. Clearly, it can also be important to prevent an adversary from deducing that a confidential event *has* occurred. This is the purpose of *BSD*, another *BSP* from the assembly kit.

For a view $\mathcal{V}$, *BSD* requires that the occurrence of an event from C does *not add* possible low-level observations. Considering the system after a trace $\beta.\langle c\rangle$ has occurred, any observation $\overline{\alpha}$ that is possible must have been possible also without $c \in C$ in the trace. Consequently, some sequence $\alpha' \in (V \cup N)^*$ must be enabled after β where α' may differ from α only in events from N. For a given view $\mathcal{V} = (V, N, C)$, $BSD_{\mathcal{V}}(Tr)$ is formally defined as follows:

$$\begin{aligned} BSD_{V,N,C}(Tr) \quad \equiv \quad & \forall \alpha, \beta \in E^*. \forall c \in C. ((\beta.\langle c\rangle.\alpha \in Tr \wedge \alpha|_C = \langle\rangle) \\ & \Longrightarrow \exists \alpha' \in E^*. (\alpha'|_V = \alpha|_V \wedge \alpha'|_C = \langle\rangle \wedge \beta.\alpha' \in Tr)) \end{aligned}$$

For other *BSPs* besides *BSI* and *BSD* and the representation of various existing information flow properties in *MAKS*, we refer to [22, 24, 26].

3 Generic Thread Pools

For distributed programming, the use of multi-threaded programming languages has become extremely popular [4]. The use of concurrent threads that operate in the same address space appears to be the adequate approach for applications that are, e.g., based on the client-server paradigm. For example, this allows one to program a file server that creates, for every incoming request, a new thread that handles this request and terminates afterwards. Compared to parallelism at the level of processes, an important advantage is that context switching is far less expensive for threads.

In this article, we assume that *threads* are sequential programs and multi-threading occurs at the level of local computation which operates on a shared memory. On the other hand, *processes* are potentially distributed such that each process has its own memory. The processes communicate by a communication network (between local computations) exchanging messages rather than using shared memory. Each process is potentially a multi-threaded program.

To model the behavior of multi-threaded processes in state-event systems is technically somewhat difficult.[7] The main difficulty is that threads communicate with each other asynchronously via shared memory, while state-event systems are based on a synchronous, message-passing-like communication paradigm (cf. Section 2.1). However, to specify processes with these formalisms is very natural because handshake-based inter-process communication is synchronous.

In this section, we demonstrate how the behavior of multi-threaded processes can be modeled using state-event systems. The proposed specification is highly generic

[7] Similar problems occur when using process algebras like CSP or CCS.

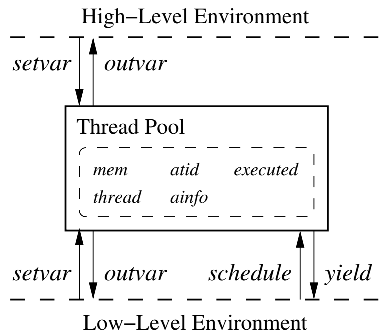

Figure 1: Generic thread pool with interface events and state objects

because it is not only parametric in the particular program but also in the programming language. How to instantiate this specification for the concrete programming language MWL will be demonstrated in Section 4. The specification will be extended to a truly distributed setting in Section 7.

3.1 Trace-Based Formal Specification

In our specification, a multi-threaded process is modeled as a collection of threads that shares a global memory. We refer to such a collection as a *thread pool*. As depicted in Figure 1, a thread pool has five state objects: *mem*, *thread*, *atid*, *ainfo*, *executed*; and can communicate with the environment by four kinds of interface events: *setvar*-, *outvar*-, *schedule*- and *yield*-events.

The *shared memory* of a thread pool is modeled by the function $mem : VAR \rightarrow VAL$ that assigns values (from VAL) to variables (from VAR). The function *mem* can be updated at the interface of a thread pool by *setvar*-events. If an event $setvar(var, val)$ occurs then variable *var* is assigned value *val*. The *outvar*-events output the value of variables to the environment. An event $outvar(var, val)$ is only enabled if *var* currently has value *val*. For simplicity, we assume that *outvar*-events have no other preconditions and do not affect the state. Moreover, *setvar*-events are always enabled.

The *setvar*-, *outvar*-, *schedule*-, and *yield*-events constitute the interface between a thread pool and its environment. For the purposes of security, direct security violations must be excluded. Hence, the environment is assumed to be separated into a high- and a low-level environment that must not communicate with each other directly. The low-level part of the environment may not inspect high-variables, i.e., occurrences of *outvar*-events on high variables must not be observable. The high-level part of the environment must not be able to alter low-level variables, i.e., occurrences of *setvar*-events on low variables must originate from the low-level part of the environment only.

The *local state of threads* is modeled by the function $thread : TID \rightarrow (THREAD \cup \{\bot, \top, \langle\rangle\})$. The function *thread* returns a local state (from $THREAD$) for a thread identifier, i.e., $thread(tid)$ denotes the local state of the thread with identifier $tid \in TID$. The results $\bot$, $\top$, and $\langle\rangle$ do not denote a proper local state but have a special meaning. If a thread with identifier *tid* has never existed then $thread(tid) = \bot$ holds (here and below, $\perp$ stands for "undefined"). After a thread has spawned child processes, the identifier of the parent thread is modified and *thread* returns, respectively, $\top$ for the old identifier and the resulting parent thread for the new identifier (which results from the old identifier by appending 0). Identifiers for child threads are chosen incrementally, i.e., the identifier of the ith child thread is constructed by appending i to the old identifier of the parent thread. If $\mathit{thread}(\mathit{tid}) = \langle\rangle$ then a thread with identifier *tid* has existed but has already terminated.

The remaining state objects are used for controlling the execution of threads. The value of $\mathit{atid} \in \mathit{TID} \cup \{\perp\}$ denotes the identifier of the thread that is currently active in the thread pool. If $\mathit{atid} = \perp$ holds then no thread is active. For simplicity, we assume that there is at most one active thread at any point of time. The state object *ainfo* is a buffer in which information is collected that shall be send to the scheduler. Note that the scheduler is external to a thread pool. The flag $\mathit{executed} \in \mathit{BOOL}$ is used for managing context switching. Thread execution proceeds as follows.

- If no thread is active (indicated by $\mathit{atid} = \perp$) then *schedule*-events are enabled. After an occurrence of $\mathit{schedule}(\mathit{tid})$, *atid* is set to *tid*, and the thread with local state $\mathit{thread}(\mathit{tid})$ becomes active. $\mathit{schedule}(\mathit{tid})$ is only enabled if the thread is alive ($\mathit{thread}(\mathit{tid}) \notin \{\perp, \top, \langle\rangle\}$).

- If there is an active thread (indicated by $\mathit{atid} \neq \perp \wedge \mathit{executed} = \mathit{ff}$) then this thread can run. Thread execution is formally modeled by the occurrence of events that are internal to the thread pool. Since these internal events depend closely on the particular instantiation of a generic thread pool, especially on the programming language, they are intentionally not modeled at the generic level. During execution, a thread can affect the state objects *mem* and *thread*. Additionally, information for the scheduler is stored in *ainfo*. Eventually, the active thread stops executing (indicated by $\mathit{executed} = \mathit{tt}$).

- After the active thread has stopped ($\mathit{executed} = \mathit{tt}$), the scheduler can be informed about this by a *yield*-event. $\mathit{yield}(\mathit{info})$ is only enabled if *info* corresponds to the actual scheduler information ($\mathit{info} = \mathit{ainfo}$). A *yield*-event resets the *executed*-flag, *atid*, and *ainfo*.

For the initial state, we assume that all variables are initialized with the value *initval*. Moreover, we assume that there is exactly one thread, which has *inittid* as identifier and *initthread* as local state. In the initial state, *atid*, *ainfo*, and *executed* are reset.

Generic thread pools are formalized as state-event systems as follows.

Definition 4 *Let VAR, VAL, TID, THREAD, and INFO be types. Let S, s_0, E_{pool}, I_{pool}, O_{pool}, and T_{pool} be defined as depicted in Figure 2.*[8] *Let $\mathit{initval} \in \mathit{VAL}$, $\mathit{inittid} \in \mathit{TID}$, $\mathit{initthread} \in \mathit{THREAD}$, E_{local} be a set of events that is disjoint from E_{pool}, and $T_{local} \subseteq S \times E_{local} \times S$ be a transition relation.*

[8] In Figure 2, the transition relation T_{pool} is specified by preconditions and postconditions. E.g., for the *setvar*-events, the precondition is trivially satisfied ($\mathit{Pre}: \ \mathit{true}$), the postcondition demands that $\mathit{mem}(\mathit{var}) = \mathit{val}$ holds *after* (indicated by primed state objects) the occurrence of $\mathit{setvar}(\mathit{var}, \mathit{val})$, and the affects-slot specifies that the values of all other state objects (except for $\mathit{mem}(\mathit{var})$) remain unchanged.

$$
\begin{array}{|l|}
\hline
S \quad = \quad \{(mem, thread, atid, ainfo, executed)\} \\
\quad \begin{array}{lcll}
mem & : & VAR \to VAL & \text{, current local shared memory} \\
thread & : & TID \to THREAD \cup \{\bot, \top, \langle\rangle\} & \text{, current local state of threads} \\
atid & : & TID \cup \{\bot\} & \text{, id of active thread} \\
ainfo & : & INFO \cup \{\bot\} & \text{, actual scheduler information} \\
executed & : & BOOL & \text{, has a step been executed?}
\end{array} \\
\hline
s_0 \quad = \quad (mem_{s_0}, thread_{s_0}, atid_{s_0}, ainfo_{s_0}, executed_{s_0}) \in S \\
\quad \forall var : VAR \, . \, mem_{s_0}(var) = initval \\
\quad thread_{s_0}(inittid) = initthread \\
\quad \forall tid : TID \, . \, tid \neq inittid \implies thread_{s_0}(tid) = \bot \\
\quad atid_{s_0} = \bot, ainfo_{s_0} = \bot, executed_{s_0} = ff \\
\hline
E_{pool} \quad = \quad I_{pool} \cup O_{pool} \\
\hline
I_{pool} \quad = \quad \{setvar(var, val), schedule(tid) \mid var : VAR, val : VAL, tid : TID\} \\
\hline
O_{pool} \quad = \quad \{outvar(var, val), yield(info) \mid var : VAR, val : VAL, info : INFO\} \\
\hline
T_{pool} \quad \subseteq \quad S \times E_{pool} \times S \\
\\
\boldsymbol{setvar}(\boldsymbol{var}, \boldsymbol{val}) \ \text{ affects } mem(var) \\
\qquad Pre: \ true \\
\qquad Post: \ mem'(var) = val \\
\boldsymbol{outvar}(\boldsymbol{var}, \boldsymbol{val}) \ \text{ affects —} \\
\qquad Pre: \ mem(var) = val \\
\qquad Post: \ true \\
\boldsymbol{schedule}(\boldsymbol{tid}) \ \text{ affects } atid \\
\qquad Pre: \ atid = \bot \wedge thread(tid) \notin \{\bot, \top, \langle\rangle\} \\
\qquad Post: \ atid' = tid \\
\boldsymbol{yield}(\boldsymbol{info}) \ \text{ affects } executed, atid, ainfo \\
\qquad Pre: \ executed = tt \wedge ainfo = info \\
\qquad Post: \ executed' = ff \wedge atid' = \bot \wedge ainfo' = \bot \\
\hline
\end{array}
$$

Figure 2: Definition of fixed components of a generic thread pool

e	side conditions	$level(e)$	$e \in$
$schedule(tid)$	$true$	low	L_{TP}
$yield(info)$	$true$	low	L_{TP}
$setvar(var, val)$	$dom_{var}(var) = low$	low	L_{TP}
$outvar(var, val)$	$dom_{var}(var) = low$	low	L_{TP}
$setvar(var, val)$	$dom_{var}(var) = high$	$high$	HI_{TP}
$outvar(var, val)$	$dom_{var}(var) = high$	$high$	$H_{TP} \backslash HI_{TP}$
e	$e \in E_{local}$	$high$	$H_{TP} \backslash HI_{TP}$

Figure 3: Security levels for events

A generic thread pool *is parametric in VAR, VAL, TID, THREAD, INFO, initval, initthread, inittid,* E_{local}*,* T_{local} *and is defined by the following state-event system:*

$$GenPool(VAR, VAL, TID, THREAD, INFO, initval, initthread, inittid, E_{local}, T_{local})$$
$$= (S, s_0, E_{pool} \cup E_{local}, I_{pool}, O_{pool}, T_{pool} \cup T_{local})$$

3.2 Security of Thread Pools

The problem of information flow control in multi-threaded programming languages is to prevent information flow from high to low variables. For this purpose, a security level (*low* or *high*) is assigned to each variable by a function $dom_{var} : var \to \{low, high\}$. This differs from the event-based approach, in which information flow control prevents the occurrence or non-occurrence of confidential events from affecting the possibility of observable behaviors. Although both approaches share the same intuitive motivation, i.e., that there should be no information flow from high to low, this technical difference complicates an integration of the two approaches. However, an integration is very desirable because it allows for a uniform investigation of information flow at the level of processes as well as at the level of threads.

The key observation, which will allow us to integrate the two approaches, is that high-level data can only be introduced into a thread pool by occurrences of *setvar*-events that change the value of high-level variables. All other events can change the state of the thread pool but cannot increase the confidentiality of data. Thus, we can express the security requirement by demanding that the occurrences of these *setvar*-events must not influence the possibility of low-level observations.

It is natural to extend security level assignments from variables to events. We denote a level-assignment function on events by $level : E_{pool} \cup E_{local} \to \{low, high\}$. We assume that a (malicious) low-level user has complete knowledge about the definition of thread pools (as usual), can observe the occurrence of *schedule*- and *yield*-events, and can observe the occurrences of *outvar*- and *setvar*-events that involve only low-level variables. Consequently, these events are assigned level *low*. All other events, i.e., *setvar*- and *outvar*-events on high variables and local events, are assigned level *high* (as displayed in Figure 3). The view $\mathcal{V}_{TP} = (L_{TP}, H_{TP} \backslash HI_{TP}, HI_{TP})$ expresses the necessary restrictions on the flow of information within a thread pool *TP* where

$$\begin{array}{rcl} L_{TP} & = & \{e \in E_{pool} \cup E_{local} \mid level(e) = low\} \\ HI_{TP} & = & \{e \in E_{pool} \cup E_{local} \mid level(e) = high\} \cap I_{pool} \\ H_{TP} \backslash HI_{TP} & = & \{e \in E_{pool} \cup E_{local} \mid level(e) = high\} \setminus I_{pool} \end{array}$$

(cf. Figure 2 for the definition of I_{pool} and E_{pool}). According to $\mathcal{V}_{TP}$, only low-level events (set L_{TP}) are visible and only high-level inputs (set HI_{TP}) are confidential. The last column of Figure 3 shows the partition of events into these classes.

Note that the assumption that the scheduler's actions (*schedule*- and *yield*-events) are low-observable adequately reflects that the scheduler is a part of the low-level environment. In particular, this model rules out all insecure schedulers, i.e., schedulers that depend on high data. The same assumption stipulates that an attacker has full knowledge of the scheduler. Indeed, even if the attacker may initially have no knowledge about the scheduler, such knowledge is possible to obtain experimentally by getting the system to execute code supplied by the attacker. Finally, yet another implication of the assumption is that the attacker is capable of observing timing behavior of the program through observing *schedule*- and *yield*-events. This allows for expressing timing-sensitive security properties.[9]

Definition 5 *The security property SecProp for thread pools is defined by:*

$$\mathit{SecProp} = (\{\mathcal{V}_{TP}\}, \mathit{BSI} \wedge \mathit{BSD})$$

A thread pool satisfies *SecProp* if *BSI* and *BSD* hold for the view $\mathcal{V}_{TP}$. Note that *BSI* (cf. Section 2.2) alone would already be an appropriate definition of information flow for this application.[10] The argument is as follows: if changing the value of high-level variables does not eliminate the possibility of low-level behaviors, then there is no information flow from high to low because high-level variables could have any value at any given point of time. Technically, the same effect can be achieved by demanding only *BSD*. The motivation for requiring both *BSI* and *BSD* will become clear in Section 7 when we extend our results to a truly distributed setting.

In general, choosing a definition of information flow closely depends on the particular application under consideration and there appears not to be a single "right" definition (as, e.g., also observed in [34]). The assembly kit offers a (still growing) collection of very primitive definitions of information flow (*BSPs*) and allows one to assemble these into more complex definitions (security predicates). This fine-grained view has proved to be very helpful for determining *SecProp*. Interestingly, *SecProp* corresponds to *generalized noninterference* [28], a well known security property. These aspects and also the compositionality of *SecProp* will be investigated in greater detail in Section 7.7.

4 MWL Thread Pools

In this section, we revisit the simple multi-threaded while-language (abbreviated by MWL) along with the timing-sensitive definition of security for MWL from [38]. Further, we demonstrate how our generic specification of thread pools from Section 3 can be instantiated for MWL.

4.1 The Multi-Threaded While-Language MWL

MWL is a shared-variable multi-threaded while-language with dynamic thread creation. The syntax of MWL commands is given by the grammar in Figure 4. As usual,

[9] It might be more appropriate to consider *schedule*- and *yield*-events as invisible in a setting where the observational power of the attacker is reduced. Technically, this can be done, e.g., by assigning security level *high* rather than *low* to these events and restricting confidential events in the view explicitly to high-level *setvar*-events rather than all high-level input events. We believe that these assignments need to be imposed for linking event-system-based security with language-based information flow in, e.g., the line of [40, 41].

[10] In [27], the *BSP BSIA* (for *backwards strict insertion of admissible confidential events*) was used as security predicate. In this article, *BSI* is used instead because the definition of *BSI* is simpler than the one of *BSIA*. Note that for thread pools, *BSI* and *BSIA* are equivalent since *setvar*-events are always admissible.

boolean expressions B range over *BOOL* and arithmetic expressions *Exp* range over *EXP*. Let $C, D, E, \ldots$ range over commands (MWL threads) *CMD*, and let $\vec{C}$ denote a vector of commands of the form $\langle C_1 \ldots C_n \rangle$. Vectors $\vec{C}, \vec{D}, \vec{E}, \ldots$ range over $\vec{CMD} = \cup_{n \in \mathbb{N}} CMD^n$, the set of multi-threaded programs.

MWL programs execute under a shared memory on a single processor (or in a single process) such that at most one thread can be active at any given point of time. A *configuration* $\langle\!| C, mem |\!\rangle$ (or $\langle\!| \vec{C}, mem |\!\rangle$) is a pair, consisting of a command $C \in CMD$ (or a vector of commands $\vec{C} \in \vec{CMD}$) and a memory $mem : VAR \rightarrow VAL$. *mem* is a finite mapping from variables to values, as in Section 3. The set of variables is partitioned into high and low security classes. For simplicity (but without loss of generality, because the case of multiple variables is handled similarly), we will assume that there is only one variable for each security class, h and l, respectively. We will often write the memory simply as a pair (val_h, val_l) with the values val_h for h and val_l for l. Further, we define *low-equivalence* on memories by: $mem_1 =_L mem_2$ if and only if the values of l for mem_1 and mem_2 are the same. The small-step semantics is given by transitions between configurations. The deterministic part of the semantics is defined by the transition rules in Figure 5. Arithmetic and boolean expressions are executed atomically by $\downarrow$ transitions. $Exp \downarrow^{mem} val$ denotes that $Exp \in EXP$ evaluates to *val* where the memory *mem* in the index is only important if *Exp* contains variables. Similarly, $B \downarrow^{mem} tt$ and $B \downarrow^{mem} f\!f$ denote, respectively, that $B \in BOOL$ evaluates to *true* or *false*.

The $\twoheadrightarrow$-transitions are deterministic. The general form of a deterministic transition is either $\langle\!| C, mem |\!\rangle \twoheadrightarrow \langle\!| \langle\rangle, mem' |\!\rangle$, which means termination with the final memory mem', or $\langle\!| C, mem |\!\rangle \twoheadrightarrow \langle\!| C'\vec{D}, mem' |\!\rangle$. Here, one step of computation starting with command C in a memory *mem* gives a new main thread C', a vector $\vec{D}$ of spawned threads, and a new memory mem'. The command $\mathsf{fork}(C\vec{D})$, where $\vec{D}$ is required to be nonempty, dynamically creates a new vector $\vec{D}$ of threads that, afterwards, run in parallel with the main thread C. This has the effect of adding the vector $\vec{D}$ to the configuration. The rule Pick in Figure 6 defines the concurrent semantics of MWL. Whenever the scheduler picks a thread C_i for execution, then a $\rightarrow$-transition takes place updating the command pool and the shared memory according to a (small) computation step of C_i. Let $\rightarrow^*$ denote the reflexive and transitive closure of $\rightarrow$.

While the rule Pick is nondeterministic, it is, in general, important to explicitly model the scheduler for addressing flows that result from scheduling policies. Yet, as we will see later, our security condition for MWL is defined purely on $\twoheadrightarrow$-transitions. As has been shown in [38], this condition implies scheduler-independent security. Thus, there is no reason to introduce explicit schedulers in the semantics for our purposes. Indeed, our main goal here is to relate the semantics and security of MWL to the corresponding event system in a possibilistic setting.

We can extract a simple model of the timing behavior of multi-threaded programs from the small-step semantics. This is done by the assumption that each $\twoheadrightarrow$-transition takes a single unit of time to execute. This approach gives only a rough approximation of real timing behavior, but simple extensions are possible in order to make it sensitive to the timing behavior of particular commands (cf. [2]).

$$
\begin{aligned}
C ::= {} & \mathsf{skip} \mid \mathit{var} := \mathit{Exp} \mid C_1 ; C_2 \mid \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \\
& \mid \mathsf{while}\ B\ \mathsf{do}\ C \mid \mathsf{fork}(C\ \vec{D})
\end{aligned}
$$

Figure 4: Command syntax

[Skip] $\langle\!|\mathsf{skip}, \mathit{mem}|\!\rangle \rightharpoonup \langle\!|\langle\rangle, \mathit{mem}|\!\rangle$

[Assign] $\dfrac{\mathit{Exp} \downarrow^{\mathit{mem}} n}{\langle\!|\mathit{var} := \mathit{Exp}, \mathit{mem}|\!\rangle \rightharpoonup \langle\!|\langle\rangle, \mathit{mem}[\mathit{var} \mapsto n]|\!\rangle}$

[Seq$_1$] $\dfrac{\langle\!|C_1, \mathit{mem}|\!\rangle \rightharpoonup \langle\!|\langle\rangle, \mathit{mem}'|\!\rangle}{\langle\!|C_1 ; C_2, \mathit{mem}|\!\rangle \rightharpoonup \langle\!|C_2, \mathit{mem}'|\!\rangle}$

[Seq$_2$] $\dfrac{\langle\!|C_1, \mathit{mem}|\!\rangle \rightharpoonup \langle\!|C_1' \vec{D}, \mathit{mem}'|\!\rangle}{\langle\!|C_1 ; C_2, \mathit{mem}|\!\rangle \rightharpoonup \langle\!|(C_1' ; C_2)\vec{D}, \mathit{mem}'|\!\rangle}$

[If$_{tt}$] $\dfrac{B \downarrow^{\mathit{mem}} \mathit{tt}}{\langle\!|\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, \mathit{mem}|\!\rangle \rightharpoonup \langle\!|C_1, \mathit{mem}|\!\rangle}$

[If$_{ff}$] $\dfrac{B \downarrow^{\mathit{mem}} \mathit{ff}}{\langle\!|\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, \mathit{mem}|\!\rangle \rightharpoonup \langle\!|C_2, \mathit{mem}|\!\rangle}$

[While$_{tt}$] $\dfrac{B \downarrow^{\mathit{mem}} \mathit{tt}}{\langle\!|\mathsf{while}\ B\ \mathsf{do}\ C, \mathit{mem}|\!\rangle \rightharpoonup \langle\!|C ; \mathsf{while}\ B\ \mathsf{do}\ C, \mathit{mem}|\!\rangle}$

[While$_{ff}$] $\dfrac{B \downarrow^{\mathit{mem}} \mathit{ff}}{\langle\!|\mathsf{while}\ B\ \mathsf{do}\ C, \mathit{mem}|\!\rangle \rightharpoonup \langle\!|\langle\rangle, \mathit{mem}|\!\rangle}$

[Fork] $\langle\!|\mathsf{fork}(C\vec{D}), \mathit{mem}|\!\rangle \rightharpoonup \langle\!|C\vec{D}, \mathit{mem}|\!\rangle$

Figure 5: Small-step deterministic semantics of commands

[Pick] $\dfrac{\langle\!|C_i, \mathit{mem}|\!\rangle \rightharpoonup \langle\!|\vec{C}, \mathit{mem}'|\!\rangle}{\langle\!|\langle C_1 \dots C_n\rangle, \mathit{mem}|\!\rangle \rightarrow \langle\!|\langle C_1 \dots C_{i-1}\vec{C}C_{i+1} \dots C_n\rangle, \mathit{mem}'|\!\rangle}$

Figure 6: Concurrent semantics of programs

4.2 Definition of Security for MWL

Now, we define the security of MWL programs and motivate the choice of this definition. The central idea of *extensional* security, as opposed to *intensional* security, is that confidentiality should not be specified by a special-purpose security formalism, but, rather, should be defined in terms of a standard semantics as a dependency property (more precisely, absence of dependence). If direct, indirect, and timing flows are considered, then, intuitively, a program has the extensional *noninterference* property, if varying the high input will not change the possible low-level observations, i.e., low inputs/outputs and timing. This differs from intensional security which relies on particular security primitives that are only motivated by intuition rather than a mathematical justification. Many investigations have successfully followed the extensional view including [8, 45, 19, 42, 1, 44, 21, 38, 2, 39] for justification of security analysis and verification techniques for different languages. We follow the extensional approach and focus on extensional security for MWL.

A previous investigation [38] gives an account on choosing an adequate definition of extensional security for multi-threaded programs. Which definition is appropriate depends on, for instance, whether a particular scheduler is assumed, or a robust scheduler-independent security is wanted. The central idea of the bisimulation-based approach is to define a *low-bisimulation* on commands such that the indistinguishability of the behaviors of two programs C and D for the attacker is formalized by $C \sim_L D$, where $\sim_L$ is a *low-bisimulation*. Such an approach is flexible in the choice of an appropriate low-bisimulation (different low-bisimulations are available for different degrees of security). For a given low-bisimulation $\sim_L$, the definition of security is simply: "C is secure iff $C \sim_L C$". Note that such a relation $\sim_L$ is not necessarily reflexive. Indeed, the intention is that insecure programs are not related by $\sim_L$ to itself. For the purpose of this article we adopt the *strong low-bisimulation* [38].

Definition 6 *Define* strong low-bisimulation $\cong_L$ *to be the union of all symmetric relations R on MWL command pools (programs) of equal size for which whenever $\langle C_1 \dots C_n \rangle \; R \; \langle D_1 \dots D_n \rangle$ then*

$$\forall mem_1, mem_2, i.(\langle\!| C_i, mem_1 |\!\rangle \twoheadrightarrow \langle\!| \vec{C'}, mem'_1 |\!\rangle \wedge \; mem_1 =_L mem_2) \Longrightarrow$$
$$(\exists \vec{D'}, mem'_2 . \langle\!| D_i, mem_2 |\!\rangle \twoheadrightarrow \langle\!| \vec{D'}, mem'_2 |\!\rangle \wedge \; mem'_1 =_L mem'_2 \wedge \vec{C'} \; R \; \vec{D'})$$

Our definition of security for MWL programs is based on strong low-bisimulation. The choice of this particular bisimulation results in a definition of security that is *timing-sensitive* and *scheduler-independent*. If two commands might have a different timing behavior depending on high data (which would result in information flow from high to low) then they are not low-bisimilar. Strong bisimulation guarantees scheduler-independent security which is robust with respect to a wide class of a schedulers (including probabilistic schedulers as shown in [38]). Although these features impose restrictions[11] on what can be considered low-bisimilar, the choice of strong

[11] For example, one restriction is the requirement that two low-bisimilar programs must have the same number of threads. However, if this requirement is lifted, secret information might be revealed to the attacker through certain schedulers (see Section 4.3 in [38]).

low-bisimulation is adequate (not too restrictive) for, e.g., the type-based analysis that is proposed in [38]. This analysis is sound with respect to the security definition, i.e., if a program passes the analysis, then it must be secure. For more details on the power of this type of security definition to capture insecure programs and examples of secure programming with common algorithms, such as sorting and searching, we refer to [38, 3].

Definition 7 *An MWL program $\vec{C}$ is* secure *if and only if $\vec{C} \approxeq_L \vec{C}$.*

In order to illustrate Definitions 6 and 7 we give some examples of secure and insecure information flow which may occur in MWL programs. Recall that a memory is a pair (val_h, val_l) of the values val_h and val_l of the variables h and l, respectively.

$l := h$ This is an example of a *direct* flow. To see that this program is insecure according to Definition 7, choose $mem_1 = (0,0)$ and $mem_2 = (1,0)$. Since $\langle\!| l := h, (0,0) |\!\rangle \twoheadrightarrow \langle\!| \langle\rangle, (0,0) |\!\rangle$ and $\langle\!| l := h, (1,0) |\!\rangle \twoheadrightarrow \langle\!| \langle\rangle, (1,1) |\!\rangle$ holds, the resulting memories are not low-equivalent $(0,0) \neq_L (1,1)$. Thus, there cannot be a relation with the properties necessary for strong low-bisimilarity.

if $h = 1$ **then** $l := 1$ **else** $l := 0$ This exemplifies an *indirect* flow through branching on a high condition. If the computation starts with low-equivalent memories $(0,0)$ and $(1,0)$, then, after one step of the computation (the test of the condition), the memories are still low-equivalent. However, after another computation step they become different depending on the initial value of h. There cannot be a relation with the properties necessary for strong low-bisimilarity.

if $h = 1$ **then** (**while** $h <$ ***MaxInt*** **do** $h := h + 1$) **else skip** From the timing behavior of the program the attacker may deduce secret information. This is an instance of a *timing* leak. Clearly, the timing behavior of the branches is different. This is captured by Definition 7. Indeed, in case the then-branch of the if is chosen, there will be no transition in the other branch to match the transitions of the while-loop.

if $h = 1$ **then** (**while** ***true*** **do skip**) **else skip** is a variation of the timing leak called a *termination* leak.

All examples above are insecure according to our definition. Here is an instance of a secure program:

if $h = 1$ **then** $h := h + 1$ **else skip** Indeed, the timing behavior is independent of the value of h, as well as the low variable l. A suitable symmetric relation that makes this program low-bisimilar to itself is, e.g., the relation $\{(\text{if } h = 1 \text{ then } h := h+1 \text{ else skip}, \text{if } h = 1 \text{ then } h := h+1 \text{ else skip}), (h := h+1, \text{skip}), (\text{skip}, h := h+1), (h := h+1, h := h+1), (\text{skip}, \text{skip}), (\langle\rangle, \langle\rangle)\}$.

4.3 Instantiating Generic Thread Pools

We now instantiate our generic model for thread pools from Section 3 in order to model the behavior of the multi-threaded programs of MWL. Recall, that, according to Definition 4, the following parameters must be actualized:

- types: *VAR*, *VAL*, *TID*, *THREAD*, *INFO*,
- initial values: *initval*, *initthread*, *inittid*,
- internal events: E_{local}; and their behavior: T_{local}.

Consistently with the simplification of Section 4.1, the set *VAR* of variables consists of only two variables h and l (having in mind that h is a high-level and l a low-level variable). We do not further specify the set *VAL* of values. However, as in the previous section, we assume that there are sets of arithmetic *EXP* and boolean *BOOL* expressions. As before, $Exp \downarrow^{mem} val$ (resp., $B \downarrow^{mem} tt$ or $B \downarrow^{mem} ff$) denote that $Exp \in EXP$ evaluates to *val* (resp., $B \in BOOL$ evaluates to *true* or *false*). *TID* is specialized to the set of sequences of natural numbers ($TID = \mathbb{N}^*$). The set *THREAD* is specialized to *CMD*, i.e., the local state of a thread is simply an MWL command. *INFO* is specialized to $VAL \times INT$ where *VAL* is the value of the priority variable (which is adapted to be l for simplicity) and the *INT* part says whether the process has been killed (value -1), continues running (value 0) or has spawned $n > 0$ new processes (value n).

We do not further specify *initval*, the initial value of all variables. The identifier of the (unique) initial thread is zero, i.e., $inittid = 0$. MWL thread pools shall be parametric in the initial thread (parameter *initthread*).

We now introduce two auxiliary functions *first* and *rest* on commands. The purpose of *first* and *rest* is to decompose sequential compositions. If a command C is not itself a sequential composition on the top level, then $first(C) = C$ and $rest(C) = \langle\rangle$. If C can be written in the form $C_1; C_2$ such that C_1 is not a sequential composition on the top level, then $first(C) = C_1$ and $rest(C) = C_2$.

The set E_{local}^{MWL} of internal events of an MWL thread pool is defined in Figure 7. Note that for each of these events there is a corresponding rule of the small-step semantics (cf. Figure 5). E.g., the *assign*-events correspond to rule Assign and the events ite^{tt} and ite^{ff} respectively correspond to If_{tt} and If_{ff}. With the exception of the rules Seq_1 and Seq_2, there are corresponding events in E_{local}^{MWL} for each rule in Figure 5. The reason for this correspondence is that, on the one hand, events model atomic actions and, on the other hand, rules of a small-step semantics model atomic transitions between states (or configurations—in the case of MWL). The atomic actions that can occur during the execution of an MWL thread pool include, taking up time (caused by skip), assignments to variables, branching in the control flow depending on boolean tests (if or while), or spawning of threads (fork). Note that, we do not consider the decomposition of sequentially composed commands as a separate action. Thus, there are no corresponding events.

The behavior of internal events is defined by the transition relation T_{local}^{MWL} (cf. Figure 7). Clearly, T_{local}^{MWL} should reflect the semantics of MWL. The pre- and postcondition of each event shall capture the corresponding rule of the small-step semantics. E.g., the precondition of $assign(var, val)$ requires that there is an active thread ($atid \neq \perp$) that has not already executed a command ($executed = ff$), the current command must be an assignment ($first(thread(atid)) = var := Exp$), and the expression *Exp* must evaluate to *val* under the current memory ($Exp \downarrow^{mem} val$). Note that, when new threads are spawned, then the generation of thread identifiers is managed in such a way that no

$$E^{\text{MWL}}_{local} = \{skip, assign(var, val), ite^{tt}(B, C_1, C_2), ite^{ff}(B, C_1, C_2)$$
$$while^{tt}(B, C_1), while^{ff}(B, C_1),\ fork(C, \vec{D}) \mid$$
$$var : VAR, val : VAL, B : BOOL, C, C_1, C_2 : CMD, \vec{D} : \vec{CMD}\}$$

$$T^{\text{MWL}}_{local} \subseteq S \times E^{\text{MWL}}_{local} \times S$$

skip affects *thread*(*atid*), *executed*, *ainfo*

$Pre:\ ready \wedge first(thread(atid)) = \mathsf{skip}$

$Post:\ thread'(atid) = rest(thread(atid)) \wedge done$
$\wedge ainfo' = (mem(l), terminates(thread(atid)))$

assign(***var***, ***val***) affects *mem*(*var*), *thread*(*atid*), *executed*, *ainfo*

$Pre:\ ready \wedge Exp \downarrow^{mem} val \wedge first(thread(atid)) = var := Exp$

$Post:\ mem'(var) = val \wedge thread'(atid) = rest(thread(atid)) \wedge done$
$\wedge ainfo' = (mem(l), terminates(thread(atid)))$

$\boldsymbol{ite^{tt}}(B, C_1, C_2)$ affects *thread*(*atid*), *executed*, *ainfo*

$Pre:\ ready \wedge B \downarrow^{mem} tt \wedge first(thread(atid)) = \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$

$Post:\ thread'(atid) = C_1; rest(thread(atid)) \wedge done \wedge ainfo' = (mem(l), 0)$

$\boldsymbol{ite^{ff}}(B, C_1, C_2)$ affects *thread*(*atid*), *executed*, *ainfo*

$Pre:\ ready \wedge B \downarrow^{mem} ff \wedge first(thread(atid)) = \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$

$Post:\ thread'(atid) = C_2; rest(thread(atid)) \wedge done \wedge ainfo' = (mem(l), 0)$

$\boldsymbol{while^{tt}}(B, C_1)$ affects *thread*(*atid*), *executed*, *ainfo*

$Pre:\ ready \wedge B \downarrow^{mem} tt \wedge first(thread(atid)) = \mathsf{while}\ B\ \mathsf{do}\ C_1$

$Post:\ thread'(atid) = C_1; \mathsf{while}\ B\ \mathsf{do}\ C_1; rest(thread(atid)) \wedge done$
$\wedge ainfo' = (mem(l), 0)$

$\boldsymbol{while^{ff}}(B, C_1)$ affects *thread*(*atid*), *executed*, *ainfo*

$Pre:\ ready \wedge B \downarrow^{mem} ff \wedge first(thread(atid)) = \mathsf{while}\ B\ \mathsf{do}\ C_1$

$Post:\ thread'(atid) = rest(thread(atid)) \wedge done$
$\wedge ainfo' = (mem(l), terminates(thread(atid)))$

fork$(C, D_1 \dots D_n)$ affects *thread*(*atid*), *thread*(*atid*.0) ... *thread*(*atid*.n), *executed*, *ainfo*

$Pre:\ ready \wedge first(thread(atid)) = \mathsf{fork}(CD_1 \dots D_n)$

$Post:\ thread'(atid) = \top \wedge thread'(atid.0) = C; rest(thread(atid))$
$\wedge \forall i : \{1, \dots, n\}. thread'(atid.i) = D_i \wedge done \wedge ainfo' = (mem(l), n)$

where the following abbreviations are used:
$ready \iff (executed = ff \wedge atid \neq \bot)$
$done \iff (executed' = tt)$
$terminates(thread(atid))$ equals -1 if $rest(thread(atid)) = \langle\rangle$ and 0 otherwise.

Figure 7: Local events E^{MWL}_{local} and transition relation T^{MWL}_{local} of an MWL thread pool

tid is used for two different threads. This is enforced by the use of $\top$ and the hierarchical identifier generation in the postcondition of *fork*. That T^{MWL}_{local} indeed reflects the semantics of MWL will be proved in Section 5.

The instantiation of generic thread pools for MWL is summarized as follows.

Definition 8 *Let VAL be some type of values, CMD be defined as depicted in Figure 4, initval* $\in$ *VAL be some value,* E^{MWL}_{local} *and* T^{MWL}_{local} *be defined as depicted in Figure 7.*

The MWL thread pool *for initthread* $\in$ *CMD results from the following instantiation of generic thread pools:*

$$MWLPool(initthread) = GenPool(\{l, h\}, VAL, \mathbb{N}^*, CMD, VAL \times INT, \\ initval, initthread, 0, E^{MWL}_{local}, T^{MWL}_{local})$$

5 Relating MWL Programs and MWL Thread Pools

The objective of our specification of MWL thread pools was to provide an adequate model of MWL programs and their behavior. Firstly, any behavior of an MWL thread pool should comply with the MWL semantics. Secondly, any behavior that complies with the MWL semantics should be possible for an MWL thread pool. That our specification, indeed, is adequate is ensured by the results presented in the current section.

Recall that the system models that, respectively, underlie MWL programs and MWL thread pools are somewhat different. The model underlying MWL programs is based on *trees of states* (to be precise, configurations). It is possible to enrich these trees with events but from the perspective of the underlying paradigm these events would be mere decorations. Since the model of computation is state-based, the natural communication paradigm is via shared memory. The system model underlying MWL thread pools is based on *sequences of events*. It is possible to enrich these sequences with states but from the perspective of the underlying model these states would be mere decorations. Since the system model is event-based, the natural communication paradigm is via message passing. These differences between the system models on which MWL programs and MWL thread pools are based somewhat complicate the proofs of the following theorems.

5.1 Adequacy of MWL Thread Pools

In order to relate the transition relation of MWL thread pools to the operational semantics of MWL programs, it is necessary to construct a translation from one syntax to the other. For this purpose, we define the function *cseq*, which translates a function $thread : TID \to (CMD \cup \{\bot, \top, \langle\rangle\})$ into a corresponding vector (sequence) of MWL commands.

Definition 9 *The function* $cseq : (TID \to (CMD \cup \{\bot, \top, \langle\rangle\})) \to \vec{CMD}$ *returns a vector of MWL commands for each function thread* $: TID \to (CMD \cup \{\bot, \top, \langle\rangle\})$. *cseq is defined by* $cseq(thread) = cseq_{aux}(0, thread)$ *where* $cseq_{aux} : TID \to (TID \to$

$(\mathit{CMD} \cup \{\perp, \top, \langle\rangle\})) \to \vec{\mathit{CMD}}$ *is defined as follows:*

$$\mathit{cseq}_{\mathit{aux}}(\mathit{tid}, \mathit{thread}) = \langle\rangle \text{ , if } \mathit{thread}(\mathit{tid}) \in \{\perp, \langle\rangle\}$$

$$\mathit{cseq}_{\mathit{aux}}(\mathit{tid}, \mathit{thread}) = \mathit{thread}(\mathit{tid}) \text{ , if } \mathit{thread}(\mathit{tid}) \in \mathit{CMD}$$

$$\mathit{cseq}_{\mathit{aux}}(\mathit{tid}, \mathit{thread}) = \mathit{cseq}_{\mathit{aux}}(\mathit{tid}.0, \mathit{thread}) \dots \mathit{cseq}_{\mathit{aux}}(\mathit{tid}.n, \mathit{thread}),$$
$$\text{if } \mathit{thread}(\mathit{tid}) = \top,\ n \in \mathbb{N} \text{ is chosen maximal such that } \mathit{thread}(\mathit{tid}.n) \neq \perp$$

For a thread with identifier *tid* that has already terminated (or has never existed), $\mathit{cseq}_{\mathit{aux}}$ returns the empty command sequence. However, if the thread is still running and has not spawned any child threads so far then $\mathit{cseq}_{\mathit{aux}}$ returns the command of that thread. Finally, if the thread has spawned child threads during its execution then the result of $\mathit{cseq}_{\mathit{aux}}$ is determined by a recursive application of $\mathit{cseq}_{\mathit{aux}}$ to the continuing parent thread (identifier $\mathit{tid}.0$) and to all child threads (identifier $\mathit{tid}.i$ with $i > 0$). In the latter case, it is exploited that thread identifiers are chosen incrementally by *fork*-events and that $\mathit{thread}(\mathit{tid}) = \langle\rangle$ holds after termination of a thread with identifier *tid*. Summarizing, $\mathit{cseq}_{\mathit{aux}}(\mathit{tid}, \mathit{thread})$ denotes the sequence consisting of the command of the thread with identifier *tid* and the commands of all threads spawned by this thread and its children. $\mathit{cseq}(\mathit{thread})$ denotes the command sequence for all threads that resulted from the initial thread.

In Theorem 2 we will show that every trace of an MWL thread pool models a behavior that complies with the semantics of MWL. We now present two lemmas that are helpful for proving Theorems 2, 4, and 5. The proofs of lemmas and theorems that are omitted in the text of this and subsequent sections are contained in the appendix. Throughout this section, we assume that $\mathit{SES} = (S, s_0, E, I, O, T)$ models an MWL thread pool, i.e., that $\mathit{SES} = \mathit{MWLPool}(\mathit{initthread})$ holds for some command $\mathit{initthread} \in \mathit{CMD}$.

Lemma 1 *If s is a reachable state of SES then one of the following is true:*

- $\mathit{executed}_s = \mathit{ff} \wedge \mathit{atid}_s \neq \perp \wedge \mathit{thread}_s(\mathit{atid}_s) \notin \{\perp, \top, \langle\rangle\}$,
- $\mathit{executed}_s = \mathit{tt} \wedge \mathit{atid}_s \neq \perp$, *or*
- $\mathit{executed}_s = \mathit{ff} \wedge \mathit{atid}_s = \perp$.

Hence, it suffices to consider these three cases when analyzing executions of *SES*.

Lemma 2 *Let s, s' be states of SES with $\mathit{executed}_s = \mathit{ff}$, $\mathit{atid}_s \neq \perp$, $\mathit{thread}_s(\mathit{atid}_s) \notin \{\perp, \top, \langle\rangle\}$, and $e \in E^{\mathit{MWL}}_{\mathit{local}}$ be an event such that $s \xrightarrow{e} s'$ holds.*

- *If* $e \neq \mathit{fork}(C, D_1 \dots D_n)$ *then*
  $\langle\!|\mathit{thread}_s(\mathit{atid}_s), \mathit{mem}_s|\!\rangle \twoheadrightarrow \langle\!|\mathit{thread}_{s'}(\mathit{atid}_{s'}), \mathit{mem}_{s'}|\!\rangle$.

- *If* $e = \mathit{fork}(C, D_1 \dots D_n)$ *then*
  $\langle\!|\mathit{thread}_s(\mathit{atid}_s), \mathit{mem}_s|\!\rangle \twoheadrightarrow \langle\!|\mathit{thread}_{s'}(\mathit{atid}_{s'}.0) \dots \mathit{thread}_{s'}(\mathit{atid}_{s'}.n), \mathit{mem}_{s'}|\!\rangle$.

Lemma 2 ensures that the occurrence of a local event in E^{MWL}_{local} corresponds to some small step in the operational semantics.

Theorem 2 *Let $s, s' \in S$ be reachable states of SES, $\gamma \in E^*$ be a sequence of events, $\vec{D}_s = cseq(thread_s)$ and $\vec{D}_{s'} = cseq(thread_{s'})$ be vectors of MWL commands. If $s \stackrel{\gamma}{\Longrightarrow} s'$ and γ contains no setvar-events then $\langle\!| \vec{D}_s, mem_s |\!\rangle \rightarrow^* \langle\!| \vec{D}_{s'}, mem_{s'} |\!\rangle$ holds.*

In Theorem 3, we will show that for every behavior that complies with the semantics of MWL, there is a trace of the corresponding MWL thread pool that models this behavior. The following lemma is helpful for proving this theorem and also Theorems 4 and 5.

Lemma 3 *Let $C' \in CMD \cup \{\langle\rangle\}$, $C'' \in CMD$, $D_1 \dots D_k \in \vec{CMD}$, and $mem' : VAR \rightarrow VAL$. Moreover, let s be a state of SES with $executed_s = ff$, $atid_s \neq \bot$, and $thread_s(atid_s) \notin \{\bot, \top, \langle\rangle\}$.*

1. *If $\langle\!| thread_s(atid_s), mem_s |\!\rangle \twoheadrightarrow \langle\!| C', mem' |\!\rangle$ then there exist $e \in E^{\text{MWL}}_{local}$ and $s' \in S$ with $s \stackrel{e}{\longrightarrow} s'$, $mem_{s'} = mem'$, $thread_{s'}(atid_s) = C'$, $atid_{s'} = atid_s$, and $executed_{s'} = tt$. Moreover, $thread_{s'}(tid) = thread_s(tid)$ holds for $tid \neq atid_s$.*

2. *If $\langle\!| thread_s(atid_s), mem_s |\!\rangle \twoheadrightarrow \langle\!| C'' D_1 \dots D_k, mem' |\!\rangle$ (with $k \geq 1$) then there exist $e \in E^{\text{MWL}}_{local}$ and $s' \in S$ with $s \stackrel{e}{\longrightarrow} s'$, $mem_{s'} = mem'$, $atid_{s'} = atid_s$, and $executed_{s'} = tt$. Moreover, $thread_{s'}(atid_s) = \top$, $thread_{s'}(atid_s.0) = C''$, $thread_{s'}(atid_s.i) = D_i$ holds for all $i \in \{1, \dots, k\}$, and $thread_{s'}(tid) = thread_s(tid)$ holds for $tid \notin \{atid_s, atid_s.0, \dots, atid_s.k\}$.*

Lemma 3 ensures that a small step in the operational semantics corresponds to the occurrence of some local event in E^{MWL}_{local}.

Theorem 3 *Let s be a reachable state of SES with $atid_s = \bot$ and $executed_s = ff$. Let $\vec{D}_s = cseq(thread_s)$, $\vec{D'} \in \vec{CMD}$, and $mem' : var \rightarrow val$. If $\langle\!| \vec{D}_s, mem_s |\!\rangle \rightarrow^* \langle\!| \vec{D'}, mem' |\!\rangle$ then there exists a sequence $\gamma : E^*$ that contains no setvar-events and a state $s' \in S$ such that $s \stackrel{\gamma}{\Longrightarrow} s'$, $mem_{s'} = mem'$, and $\vec{D'} = cseq(thread_{s'})$.*

Theorems 2 and 3 ensure that MWL thread pools are an adequate specification of MWL programs and their behavior. All behaviors of an MWL thread pool comply with the semantics of MWL and all behaviors that comply with the semantics of MWL are possible for an MWL thread pool.

6 Soundness and Completeness Results

The aim of this section is to establish the soundness and completeness results. Soundness means that if C is secure as an MWL program then its translation is secure as a state-event system. On the other hand, completeness means that if C's translation is secure as a state-event system then C is secure.

Recall that, according to Definition 8 from Section 4, the translation $MWLPool(C)$ of an MWL program C is a thread pool with $initthread = C$. The following two subsections present the soundness and completeness results, respectively.

6.1 Soundness

Before we present the soundness theorem we state a security invariance lemma. Intuitively, the lemma says that if computation starts with a secure program then all the threads in the thread pool are secure at all times. As an abbreviation, we define the predicate $\mathit{live}(s, \mathit{tid}) \iff (\mathit{thread}_s(\mathit{tid}) \notin \{\bot, \top, \langle\rangle\})$ that holds whenever thread *tid* is alive in state s (exists and has not terminated). Note that $\mathit{live}(s, \mathit{tid})$ is a precondition for scheduling thread *tid* in s.

Lemma 4 *Assume an MWL program C is secure and $\beta \in \mathit{Tr}$ is a trace for MWLPool(C) such that $s_0 \xRightarrow{\beta} s$. Then $\forall \mathit{tid}.\ \mathit{live}(s, \mathit{tid}) \implies \mathit{thread}_s(\mathit{tid}) \cong_L \mathit{thread}_s(\mathit{tid})$.*

Theorem 4 (Soundness) *If an MWL program C is secure then the MWL thread pool MWLPool(C) satisfies the security property SecProp.*

6.2 Completeness

Let us first recall some facts from standard bisimulation theory before we turn to proving completeness. Restating Definition 6, two thread pools $\vec{C} = \langle C_1 \dots C_n \rangle$ and $\vec{D} = \langle D_1 \dots D_n \rangle$ are strongly low-bisimilar $\vec{C} \cong_L \vec{D}$ iff $\exists R.\, R \subseteq F(R)$ where function F from pers to pers (partial equivalence relations over $\vec{\mathit{CMD}}$) is given by: $\vec{C}\ F(R)\ \vec{D}$ iff

$$\begin{aligned}&\forall \mathit{mem}_1, \mathit{mem}_2, i.(\langle\!| C_i, \mathit{mem}_1 |\!\rangle \twoheadrightarrow \langle\!| \vec{C'}, \mathit{mem}_1' |\!\rangle \wedge\ \mathit{mem}_1 =_L \mathit{mem}_2) \implies \\ &\quad (\exists \vec{D'}, \mathit{mem}_2'.\langle\!| D_i, \mathit{mem}_2 |\!\rangle \twoheadrightarrow \langle\!| \vec{D'}, \mathit{mem}_2' |\!\rangle \wedge\ \mathit{mem}_1' =_L \mathit{mem}_2' \wedge \vec{C'}\ R\ \vec{D'})\end{aligned}$$

Let us state two lemmas that give an alternative representation for the strong low-bisimulation. The proof of the lemmas is a standard argument, by appeal to the Knaster-Tarski fixed-point theorem (see, e.g., [9]).

Lemma 5 *Function F is ω-cocontinuous, i.e., for a nonincreasing ω-chain of pers $R_0 \supseteq \dots \supseteq R_i \supseteq \dots$, F preserves colimits:*

$$F(\cap_{i<\omega} R_i) = \cap_{i<\omega} F(R_i)$$

Lemma 6 (Fixed point) *The relation $\cong_L$ is the greatest fixed point of F in the lattice of pers. It can be alternatively represented by $\cong_L = \cap_{i<\omega} \cong_L^i$ where $\cong_L^{i+1} = F(\cong_L^i)$ and $\cong_L^0$ is the total relation $\vec{\mathit{CMD}} \times \vec{\mathit{CMD}}$.*

We are now ready to present the completeness result.

Theorem 5 (Completeness) *An MWL program C is secure whenever MWLPool(C) satisfies the security property SecProp.*

7 Secure Communication for Distributed Programs

So far, we have only considered local multi-threaded computation based on MWL. Our original motivation was to embrace the security of both local computations and communication between local computations. Thus, the next step is to investigate system security in the case of distributed programming. This section presents DMWL (distributed MWL), which is an extension of MWL with message-passing primitives for distributed programming. In this section, we define the security of DMWL programs, translate DMWL's semantics into an event-based setting, show the adequacy of the translation and give soundness and completeness results. The section culminates with presenting a compositionality result that allows for decentralized compositional design of systems that fulfill global security.

7.1 DMWL's Communication Primitives

In a distributed setting each process has its own memory. Thus, the processes communicate by a communication network exchanging messages rather than using shared memory. Typical examples of message-passing-based distributed implementations are client-server applications. Recall the file server example from the beginning of Section 3. The file server program creates a new thread for every incoming request, and terminates afterwards. Such a request is nothing else but a message passed by a client program that, e.g, needs to open, read, write and then close a file. The server grants read and write permissions by sending respective messages to clients. Message passing for distributed programming has been adopted by many distributed languages including Erlang [5], Java [16] (message-passing primitives are available in the standard `java.net` package), and Linda [6]. In fact, any sequential language can be augmented with Linda, which is a collection of message-passing primitives implemented on the top of the sequential language using tagged tuples.

Message passing is based on sending and receiving messages on *channels*, which can be thought of as *links* between processes. We assume that each link connects two processes in one direction and that no process can be linked to itself. Each channel is a FIFO queue of messages. Messages can be put into a channel by send commands and be taken out of a channel by receive commands.

In this section, we extend MWL with such communication primitives. The new language is called DMWL, which stands for distributed MWL. Figure 8 gives DMWL's new commands apart from those of MWL. The command $\mathsf{send}(cid, Exp)$ is used for sending the result of evaluating the expression *Exp* on channel *cid*. We distinguish between two receiving primitives. The first one is a *blocking* $\mathsf{receive}(cid, var)$ which blocks until it receives a value on the channel *cid*. Once the value has been received, the variable *var* is set to that value. The *nonblocking* receive $\mathsf{if\text{-}receive}(cid, var, C_1, C_2)$ always continues execution. If the channel *cid* is nonempty then *var* is set to the received value and execution continues with the command C_1. Otherwise execution continues with the command C_2.

Figure 9 defines the deterministic semantics of these commands. Deterministic transitions between configurations are denoted, as before, by $\twoheadrightarrow$-arrows. Now a configuration has the form $\langle\!|C, mem, \sigma|\!\rangle$ where the difference to an MWL configuration is the *channel status* function $\sigma : \mathit{CID} \to \mathit{VAL}^*$. Given *cid* this function returns the queue of messages that are currently waiting on channel *cid*. The deterministic transitions for MWL, defined in Figure 5, are part of the DMWL semantics with the modification that configurations are extended with σ. The channel status function σ remains unchanged in those transitions. The same extension is made for the rule Pick for nondeterministic transitions (denoted by $\rightarrowtriangle$-arrows). The extended rule is depicted in Figure 10.

Given a (possibly distributed) collection of DMWL programs $\vec{C}_1, \ldots, \vec{C}_n$, in which each program executes on its own memory $\mathit{mem}_1, \ldots, \mathit{mem}_n$, respectively, a *global* DMWL configuration $\lhd(\vec{C}_1, \mathit{mem}_1), \ldots, (\vec{C}_n, \mathit{mem}_n); \sigma \rhd$ consists of two components. The first component is a sequence of pairs each containing a DMWL program and its memory. The second component is the current channel status function σ. New nondeterministic $\twoheadrightarrow$-transitions on global configurations are defined by the rule Step in Figure 11. The rule Step is similar to the rule Pick. It ensures that a global transition takes place whenever a local transition occurs.

7.2 Security of DMWL Programs

We assume that communication channels are partitioned into low and high channels. *Low* channels are observable by the attacker. Communication on low channels corresponds to, e.g., communication using standard Internet protocols such as TCP/IP and HTTP. Here, the traffic is vulnerable to eavesdropping by the attacker. *High* channels are secure links between processes that are invisible for the attacker. Communication on high channels corresponds to, e.g., communication within a protected Intranet, which is an IP-based network of nodes behind a firewall or behind several firewalls. Here, the traffic cannot be seen by the attacker.

Let $\mathit{dom}_{ch} : \mathit{CID} \to \{\mathit{high}, \mathit{low}\}$ be a function that given a channel id *cid* returns the security level $\mathit{dom}_{ch}(\mathit{cid})$ of that channel. Let us extend low-equality $=_L$ (defined in Section 4.1) to relate channel status functions that agree on their low arguments. Formally, define for $\sigma_1, \sigma_2 : \mathit{CID} \to \mathit{VAL}^*$:

$$\sigma_1 =_L \sigma_2 \iff (\forall \mathit{cid} \in \mathit{CID}.\ \mathit{dom}_{ch}(\mathit{cid}) = \mathit{low} \Longrightarrow \sigma_1(\mathit{cid}) = \sigma_2(\mathit{cid}))$$

In a similar way as we extended MWL semantics to DMWL semantics, we now extend strong low-bisimulation for MWL programs (Definition 6) to strong low-bisimulation for DMWL programs.

Definition 10 *Define* strong low-bisimulation $\cong_L$ *to be the union of all symmetric relations* R *on DMWL command pools (programs) of equal size for which whenever* $\langle C_1 \ldots C_n\rangle\ R\ \langle D_1 \ldots D_n\rangle$ *then*

$$\begin{array}{l} \forall \mathit{mem}_1, \mathit{mem}_2, \sigma_1, \sigma_2, i. \langle\!| C_i, \mathit{mem}_1, \sigma_1 |\!\rangle \rightarrowtriangle \langle\!| \vec{C'}, \mathit{mem}'_1, \sigma'_1 |\!\rangle \\ \qquad\qquad \wedge\ \mathit{mem}_1 =_L \mathit{mem}_2 \wedge\ \sigma_1 =_L \sigma_2 \Longrightarrow \\ \quad \exists \vec{D'}, \mathit{mem}'_2, \sigma'_2. \langle\!| D_i, \mathit{mem}_2, \sigma_2 |\!\rangle \rightarrowtriangle \langle\!| \vec{D'}, \mathit{mem}'_2, \sigma'_2 |\!\rangle \\ \qquad\qquad \wedge\ \mathit{mem}'_1 =_L \mathit{mem}'_2 \wedge \sigma'_1 =_L \sigma'_2 \wedge \vec{C'}\ R\ \vec{D'} \end{array}$$

$$C ::= \ldots \mid \mathsf{send}(cid, Exp) \mid \mathsf{receive}(cid, var) \mid \mathsf{if\text{-}receive}(cid, var, C_1, C_2)$$

Figure 8: Command syntax

$$[\text{Send}] \quad \frac{Exp \downarrow^{mem} val}{\langle\!| \mathsf{send}(cid, Exp), mem, \sigma |\!\rangle \rightharpoonup \langle\!| \langle\rangle, mem, \sigma[cid \mapsto val.\sigma(cid)] |\!\rangle}$$

$$[\text{Receive}] \quad \frac{\sigma(cid) = vals.val}{\langle\!| \mathsf{receive}(cid, var), mem, \sigma |\!\rangle \rightharpoonup \langle\!| \langle\rangle, mem[var \mapsto val], \sigma[cid \mapsto vals] |\!\rangle}$$

$$[\text{IfRcv}_{ff}] \quad \frac{\sigma(cid) = \langle\rangle}{\langle\!| \mathsf{if\text{-}receive}(cid, var, C_1, C_2), mem, \sigma |\!\rangle \rightharpoonup \langle\!| C_2, mem, \sigma |\!\rangle}$$

$$[\text{IfRcv}_{tt}] \quad \frac{\sigma(cid) = vals.val}{\langle\!| \mathsf{if\text{-}receive}(cid, var, C_1, C_2), mem, \sigma |\!\rangle \rightharpoonup \langle\!| C_1, mem[var \mapsto val], \sigma[cid \mapsto vals] |\!\rangle}$$

Figure 9: Small-step deterministic semantics of communication primitives

$$[\text{Pick}] \quad \frac{\langle\!| C_i, mem, \sigma |\!\rangle \rightharpoonup \langle\!| \vec{C}, mem', \sigma' |\!\rangle}{\langle\!| \langle C_1 \ldots C_n \rangle, mem, \sigma |\!\rangle \rightarrow \langle\!| \langle C_1 \ldots C_{i-1} \vec{C} C_{i+1} \ldots C_n \rangle, mem', \sigma' |\!\rangle}$$

Figure 10: Concurrent semantics of programs

$$[\text{Step}] \quad \frac{\langle\!| \vec{C}_k, mem_k, \sigma |\!\rangle \rightarrow \langle\!| \vec{C}'_k, mem'_k, \sigma' |\!\rangle}{\lhd (\vec{C}_1, mem_1), \ldots, (\vec{C}_n, mem_n); \sigma \rhd \twoheadrightarrow \lhd (\vec{C}_1, mem_1), \ldots, (\vec{C}'_k, mem'_k), \ldots, (\vec{C}_n, mem_n); \sigma' \rhd}$$

Figure 11: Global concurrent semantics

The channel status functions σ_1 and σ_2 are treated in the same way as memories mem_1 and mem_2. This corresponds to our assumption that high channels are not visible to the attacker whereas low channels are fully observable. A similar approach has been applied to define program security in a version of MWL enriched with high and low synchronization [35]. This definition of low-bisimulation corresponds to whether two DMWL programs are observationally equivalent in the low-level observer's point of view. The DMWL security definition is based on this representation of the attacker's view (as before in Definition 7).

Definition 11 *A DMWL program $\vec{C}$ is* secure *if and only if $\vec{C} \approxeq_L \vec{C}$.*

Having defined security for a single DMWL program, we have not yet given a security definition for an overall system that is distributed and could consist of a number of DMWL programs. In other words, having defined *local* security we have not yet defined any notion of *global* security. As we have argued, the underlying semantic model for the global distributed computation is event-based. It is also in terms of event-based systems that we will specify the global security condition for distributed programs. Rather than defining a global condition for distributed DMWL programs in an ad-hoc manner, our goal is to derive this condition. Moreover, we aim at a compositionality principle of the following flavor:

> *Proving local DMWL security for individual DMWL commands is enough for the global security of the overall (potentially distributed) system of communicating DMWL programs.*

How to exploit this principle is illustrated in Figure 12. Suppose we want to prove that two programs $\langle C_1 C_2 \rangle$ and $\langle D_1 D_2 D_3 \rangle$ constitute a secure distributed system. Proving that each of these two programs is secure relies on proving the security of each command separately according to Definition 11. A popular approach to ensuring the security of individual commands is by using security-type systems (e.g., [45, 19, 42, 1, 44, 2, 38, 39, 35]). We have proposed a sound security-type system for DMWL in a separate article [36]. That each command is secure is depicted by single-lined ovals around the commands. That each multi-threaded program is secure is illustrated by double-lined ovals around the programs. Once we have proved that the two programs are secure the soundness and completeness results of Section 7.6 assure that the corresponding state-event systems SES^1 and SES^2 satisfy the security property *SecProp*. The solid arrows in Figure 12 symbolize *(i)* the adequacy of the semantics of the programs and the state-event systems (to be shown in Section 7.5) and also *(ii)* the connection between their security (soundness and completeness to be shown in Section 7.6). The bold ovals correspond to the security of state-event systems. Finally, the compositionality results (to be presented in Section 7.7) imply that the global (trace-based) security condition holds for the overall system. Such a global system is composed of local state-event systems by the composition defined in Section 2. This corresponds to the bold oval in Figure 12 around the overall composed system.

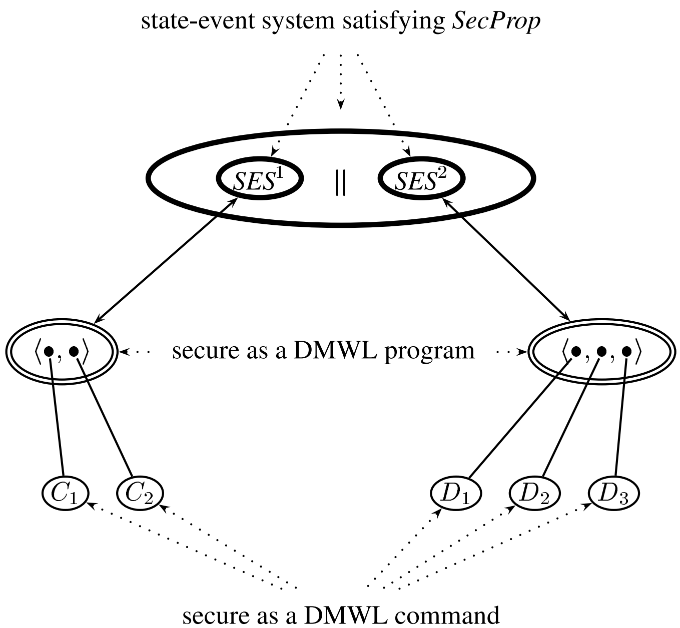

Figure 12: Derivation of global security

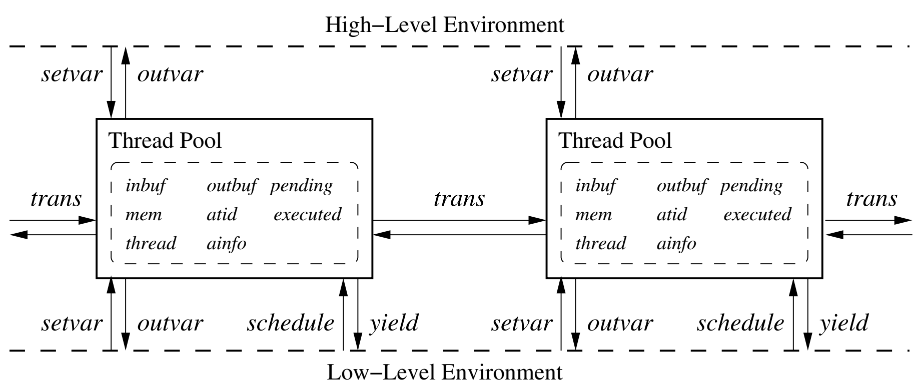

Figure 13: Generic processes with interface events and state objects

7.3 Generic Process Pools

This section describes the extension of generic thread pools (introduced in Section 3) with communication. Further, we define *generic process pools* as a composition (cf. Definition 2) of extended generic thread pools. Each thread pool corresponds to a multi-threaded process that is identified by a unique process identifier (typically denoted by *pid*). Throughout the rest of the article we use "process" synonymously to "thread pool".

Figure 13 depicts two processes (or thread pools) with new interface events *trans* along with new state objects *inbuf*, *pending* and *outbuf*. The *input buffer* state object $inbuf : CID \rightarrow VAL^*$ is a buffer for incoming messages. Given a channel id *cid*, $inbuf(cid)$ stores the sequence of messages that have been sent to the process but are not yet ready to be processed. The function $pending : CID \rightarrow VAL^*$ serves similar purposes. Namely, it stores messages that have not yet been consumed. The *output buffer* state object $outbuf : (CID \times VAL) \cup \{\langle\rangle\}$ is the dual to *inbuf*. Either *outbuf* is empty or contains a pair (cid, val) of a channel id and a value to be sent on *cid*. The actual sending is delayed until the next outgoing transmission event. Initially, these three parameters store the empty sequence for all *cid*.

A process can communicate with the environment by four kinds of interface events (*setvar*-, *outvar*-, *schedule*- and *yield*-events as before). A process can communicate with other processes by means of *transmission* events $trans(cid, val)$. Whether transmission events are input or output events to a given process is specified by functions $sender, receiver : CID \rightarrow PID$. For a channel id *cid*, $sender(cid)$ is the pid of the sender process and $receiver(cid)$ is the pid of the receiver process. We assume that *sender* and *receiver* are fixed and $\forall cid \in CID.\, sender(cid) \neq receiver(cid)$. Such functions guarantee that channels are directed links. I.e., for each channel there is exactly one sender and one receiver process and no channel can connect a process to itself.

The *setvar*, *outvar* events are defined as before. The incoming transmission events $trans(cid, val)$ (when $receiver(cid) = pid$) are always enabled. The outgoing transmission events $trans(cid, val)$ (when $sender(cid) = pid$) are enabled when the process has produced a message *val* in the output buffer value on the channel *cid*, i.e., $outbuf = (cid, val)$.

The precondition of events $schedule(tid)$ is extended by the condition that the thread with the identifier *tid* is not blocked. Essentially, a thread is blocked if it tries to receive on an empty channel. The set of blocked thread ids is abbreviated by *blocked-set* defined at the bottom of Figure 15. The postcondition of *schedule*-events ensures that values in *inbuf* are moved to *pending*. Once values are in *pending*, they are ready for processing.

yield-events have now an additional parameter *blocked-info* which propagates information about blocked threads to the scheduler. This is accounted in the precondition of *yield*: $blocked\text{-}info = blocked\text{-}set$. We also require $outbuf = \langle\rangle$, which ensures that all messages from *outbuf* have been sent out before yielding.

Generic processes are formalized as state-event systems in the following definition.

Definition 12 *Let VAR, VAL, TID, THREAD, INFO, PID, and CID be types and pid* $\in$ *PID. Let* S*,* s_0*,* E_{pool}*,* I_{pool}*,* O_{pool}*, and* T_{pool} *be defined as depicted in Figure 2 with* *the extensions depicted in Figure 15. Let initval* $\in$ VAL*, inittid* $\in$ TID*, initthread* $\in$ THREAD*,* E_{local} *be a set of events that is disjoint from* E_{pool}*,* $T_{local} \subseteq S \times E_{local} \times S$ *be a transition relation, and sender, receiver* : CID $\rightarrow$ PID *be two functions such that for all cid* $\in$ CID *holds sender(cid)* $\neq$ *receiver(cid). The* generic process *is defined by the following state-event system:*

$$\begin{aligned}&\mathit{GenProcess}(\mathit{VAR}, \mathit{VAL}, \mathit{TID}, \mathit{THREAD}, \mathit{INFO}, \mathit{PID}, \mathit{CID}, \mathit{pid},\\&\qquad\qquad \mathit{initval}, \mathit{initthread}, \mathit{inittid}, E_{local}, T_{local}, \mathit{sender}, \mathit{receiver})\\&= (S, s_0, E_{pool} \cup E_{local}, I_{pool}, O_{pool}, T_{pool} \cup T_{local})\end{aligned}$$

In order to state the security definition for generic processes we need to define the security levels for the new *trans*-events. This is done in Figure 14. Note that neither the security predicate $\mathit{BSI} \wedge \mathit{BSD}$ nor the definition of the view $\mathcal{V}_{TP}$ need to be changed (except for the addition of *trans*-events).

Definition 13 *The security property SecProp for generic processes is defined by:*

$$\mathit{SecProp} = (\{\mathcal{V}_{TP}\}, \mathit{BSI} \wedge \mathit{BSD})$$

Note that according to the view $\mathcal{V}_{TP}$, only *setvar*-events on high variables and incoming *trans*-events on high channels are confidential. I.e., for a process with identifier *pid*, an event $\mathit{trans}(\mathit{cid}, \mathit{val})$ is only confidential if $\mathit{dom}_{ch}(\mathit{cid}) = \mathit{high}$ and $\mathit{receiver}(\mathit{cid}) = \mathit{pid}$ hold.

In a distributed setting, we have a collection of generic processes such that each process has its unique process id. Such a collection forms a *generic process pool*, which we will define using the composition (Definition 2) on generic processes. Recall that in such a composition, synchronization is performed through shared events. Naturally, *trans*-events should be shared between communicating processes whereas all other events should be disjoint. To avoid cluttering the notation, we simply assume that *all but trans-events are implicitly tagged with the respective process identifiers.*

Definition 14 *Given a collection* $\mathit{GenProcess}^{pid_1}, \ldots, \mathit{GenProcess}^{pid_n}$ *of generic processes, assume that each process has the same parameters* VAL,PID,CID,*sender and receiver. Let us define a* generic process pool *by:*

$$\begin{aligned}&\mathit{GenProcPool}(\mathit{GenProcess}^{pid_1}, \ldots, \mathit{GenProcess}^{pid_n})\\&= \mathit{GenProcess}^{pid_1} \,||\cdots||\, \mathit{GenProcess}^{pid_n}\end{aligned}$$

The view for a process pool can be easily constructed from the views of the individual processes. Since dom_{ch}, *sender*, and *receiver* are defined globally, it is guaranteed that low (high) output channels may only be connected to (high) low input channels, respectively. Like for individual processes, all low-level events are assumed to be visible and all high-level input events are assumed to be confidential. When composing state-event systems, the communication events between processes become internal events of the composed system (cf. Definition 2). I.e., an event $\mathit{trans}(\mathit{cid}, \mathit{val})$ is an internal event

e	side conditions	$level(e)$	$e \in$
$trans(cid, val)$	$dom_{ch}(cid) = low$	low	L_{TP}
$trans(cid, val)$	$dom_{ch}(cid) = high \wedge receiver(cid) = pid$	$high$	HI_{TP}
$trans(cid, val)$	$dom_{ch}(cid) = high \wedge sender(cid) = pid$	$high$	$H_{TP} \backslash HI_{TP}$

Figure 14: Security levels of *trans*-events

$S \quad = \quad \{(\dots, inbuf, pending, outbuf)\}$

$inbuf$	:	$CID \rightarrow VAL^*$	, buffer for incoming messages
$pending$	:	$CID \rightarrow VAL^*$	, buffer for incoming messages
$outbuf$	:	$(CID \times VAL) \cup \{\langle\rangle\}$	, buffer for outgoing messages

$s_0 \quad = \quad (\dots, inbuf_{s_0}, pending_{s_0}, outbuf_{s_0}) \in S$

$\forall cid : CID.\, inbuf_{s_0}(cid) = \langle\rangle$
$\forall cid : CID.\, pending_{s_0}(cid) = \langle\rangle$
$outbuf_{s_0} = \langle\rangle$

$E_{pool} \quad = \quad I_{pool} \cup O_{pool}$

$I_{pool} \quad = \quad \{\dots, trans(cid, val) \mid cid : CID, val : VAL, receiver(cid) = pid\}$

$O_{pool} \quad = \quad \{\dots, trans(cid, val) \mid cid : CID, val : VAL, sender(cid) = pid\}$

$T_{pool} \quad \subseteq \quad S \times E_{pool} \times S$

$\boldsymbol{setvar(var, val)}$ affects $mem(var)$
  $Pre:\ true$
  $Post:\ mem'(var) = val$

$\boldsymbol{outvar(var, val)}$ affects —
  $Pre:\ mem(var) = val$
  $Post:\ true$

$\boldsymbol{schedule(tid)}$ affects $atid$, $inbuf$, $pending$
  $Pre:\ atid = \bot \wedge thread(tid) \notin \{\bot, \top, \langle\rangle\} \wedge tid \notin blocked\text{-}set$
  $Post:\ atid' = tid \wedge inbuf' = \langle\rangle$
    $\wedge\ \forall cid : CID.pending'(cid) = inbuf(cid).pending(cid)$

$\boldsymbol{yield(info, blocked\text{-}info)}$ affects $executed$, $atid$, $ainfo$
  $Pre:\ executed = tt \wedge ainfo = info \wedge outbuf = \langle\rangle$
    $\wedge\ blocked\text{-}info = blocked\text{-}set$
  $Post:\ executed' = ff \wedge atid' = \bot \wedge ainfo' = \bot$

$\boldsymbol{trans(cid, val)}$ (case $receiver(cid) = pid$) affects $inbuf(cid)$
  $Pre:\ true$
  $Post:\ inbuf'(cid) = val.inbuf(cid)$

$\boldsymbol{trans(cid, val)}$ (case $sender(cid) = pid$) affects $outbuf$
  $Pre:\ outbuf = (cid, val)$
  $Post:\ outbuf' = \langle\rangle$

where
$blocked\text{-}set = \{tid \mid first(thread(tid)) = \mathsf{receive}(cid, var) \wedge pending(cid) = \langle\rangle\}$

Figure 15: Definition of fixed components of a generic process with id *pid*

of the process pool if *sender*(cid) and *receiver*(cid) are identifiers of processes that are contained in the process pool. High-level *trans*-events that are internal to the process pool cannot introduce new secrets from outside the system. Consequently, these events need not be considered as confidential. This is formalized in the following definition of security for process pools.

Definition 15 *Given a process pool GenProcPool*$(GenProcess^{pid_1}, \ldots, GenProcess^{pid_n})$ *and the view* $\mathcal{V}_{PP} = (L_{PP}, H_{PP} \backslash HI_{PP}, HI_{PP})$ *defined as follows (*E_{PP} *and* I_{PP} *shall be the sets of all events and of all input events, respectively, of the process pool)*

$$\begin{array}{rcl} L_{PP} & = & \{e \in E_{PP} \mid level(e) = low\} \\ HI_{PP} & = & \{e \in E_{PP} \mid level(e) = high\} \cap I_{PP} \\ H_{PP} \backslash HI_{PP} & = & \{e \in E_{PP} \mid level(e) = high\} \setminus I_{PP} \end{array}$$

the security property SecProp for the process pool is defined by:

$$SecProp = (\{\mathcal{V}_{PP}\}, BSI \wedge BSD)$$

7.4 DMWL Process Pools

We now instantiate our generic model for process pools from Section 7.3 in order to model the behavior of DMWL programs. This section parallels and extends Section 4.3. The types and values to instantiate, according to Definition 14, are:

- types: *VAR*, *VAL*, *TID*, *THREAD*, *INFO*, *PID*, *CID*;
- initial values: *pid*, *initval*, *initthread*, *inittid*;
- connecting functions: *sender*, *receiver*;
- internal events: E_{local}; and their behavior: T_{local}.

Similarly to Section 4.3, we set $VAR = \{h, l\}$, $THREAD = CMD$ (where *CMD* is defined in Figures 4 and 8) , $inittid = 0$ and leave *VAL* and *initval* unspecified. Moreover, we do not further specify the new parameters *PID*, *CID*, *pid* and functions *sender*, *receiver*. Local events E_{local} and transition relation T_{local} are extended with communication events in the following way.

The set E_{local}^{DMWL} of internal events of a DMWL process is defined in Figures 7 and 16. For each of these events there is a corresponding rule of the small-step semantics (cf. Figures 5 and 9). The behavior of internal events is defined by the transition relation T_{local}^{DMWL} (cf. Figures 7 and 16). That T_{local}^{DMWL} indeed reflects the semantics of DMWL will be proved in Section 7.5. The instantiation of generic processes for DMWL is summarized in the following definition.

Definition 16 *The* DMWL process *for* $pid \in PID$ *and* $initthread \in CMD$ *results from the following instantiation of a generic process:*

$$\begin{array}{l} DMWLProcess(pid, initthread) \\ = GenProcess(\{l, h\}, VAL, \mathbb{N}^*, CMD, VAL \times INT, PID, CID, pid, \\ \qquad\qquad initval, initthread, 0, E_{local}^{DMWL}, T_{local}^{DMWL}, sender, receiver) \end{array}$$

$$
\begin{array}{rcl}
E^{\text{DMWL}}_{local} & = & E^{\text{MWL}}_{local} \\
 & \cup & \{ send(cid, val), receive(cid', var, val), \\
 & & \quad ite\text{-}rcv^{tt}(cid', var, val, C_1, C_2), ite\text{-}rcv^{ff}(cid', var, val, C_1, C_2) \mid \\
 & & \quad val \in VAL, var \in VAR, cid, cid' \in CID, C_1, C_2 \in CMD, \\
 & & \quad sender(cid) = pid, receiver(cid') = pid \}
\end{array}
$$

$$
T^{\text{DMWL}}_{local} \subseteq S \times E^{\text{DMWL}}_{local} \times S \qquad [T^{\text{DMWL}}_{local} \supseteq T^{\text{MWL}}_{local}]
$$

$\boldsymbol{send(cid, val)}$ affects $thread(atid)$, $executed$, $ainfo$, $outbuf$

$$
\begin{array}{ll}
Pre: & ready \wedge Exp \downarrow^{mem} val \wedge first(thread(atid)) = \mathsf{send}(cid, Exp) \\
Post: & outbuf' = (cid, val) \wedge thread'(atid) = rest(thread(atid)) \wedge done \\
 & \wedge ainfo' = (mem(l), terminates(thread(atid)))
\end{array}
$$

$\boldsymbol{receive(cid, var, val)}$ affects $thread(atid)$, $executed$, $ainfo$, $mem(var)$, $pending(cid)$

$$
\begin{array}{ll}
Pre: & ready \wedge pending(cid) \neq \langle\rangle \wedge last(pending(cid)) = val \\
 & \wedge first(thread(atid)) = \mathsf{receive}(cid, var) \\
Post: & mem'(var) = val \wedge pending'(cid) = butlast(pending(cid)) \\
 & \wedge thread'(atid) = rest(thread(atid)) \wedge done \\
 & \wedge ainfo' = (mem(l), terminates(thread(atid)))
\end{array}
$$

$\boldsymbol{ite\text{-}rcv^{tt}(cid, var, val}, C_1, C_2)$ affects $thread(atid)$, $executed$, $ainfo$, $mem(var)$, $pending(cid)$

$$
\begin{array}{ll}
Pre: & ready \wedge pending(cid) \neq \langle\rangle \wedge last(pending(cid)) = val \\
 & \wedge first(thread(atid)) = \mathsf{if\text{-}receive}(cid, var, C_1, C_2) \\
Post: & mem'(var) = val \wedge pending'(cid) = butlast(pending(cid)) \\
 & \wedge thread'(atid) = C_1; rest(thread(atid)) \wedge done \wedge ainfo' = (mem(l), 0)
\end{array}
$$

$\boldsymbol{ite\text{-}rcv^{ff}(cid, var, val}, C_1, C_2)$ affects $thread(atid)$, $executed$, $ainfo$

$$
\begin{array}{ll}
Pre: & ready \wedge pending(cid) = \langle\rangle \\
 & \wedge first(thread(atid)) = \mathsf{if\text{-}receive}(cid, var, C_1, C_2) \\
Post: & thread'(atid) = C_2; rest(thread(atid)) \wedge done \wedge ainfo' = (mem(l), 0)
\end{array}
$$

where the following abbreviations are used:
$ready \Longleftrightarrow (executed = ff \wedge atid \neq \bot)$
$done \Longleftrightarrow (executed' = tt)$
$terminates(thread(atid))$ equals -1 if $rest(thread(atid)) = \langle\rangle$ and 0 otherwise.

Figure 16: Local events E^{DMWL}_{local} and transition relation T^{DMWL}_{local} of a DMWL process

We finally arrive at a formalization of a *DMWL process pool*, which corresponds to a distributed collection of DMWL programs.

Definition 17 *Given a collection of process ids $pid_1, \ldots, pid_n \in PID$ and a collection of DMWL commands $initthread_1 \in CMD, \ldots, initthread_n \in CMD$, define a* DMWL process pool *by:*

$$\begin{aligned}&DMWLProcPool((pid_1, initthread_1), \ldots, (pid_n, initthread_n))\\&\quad = GenProcPool(DMWLProcess(pid_1, initthread_1),\\&\qquad\qquad \ldots, DMWLProcess(pid_n, initthread_n))\end{aligned}$$

Note that this definition is based on a correct use of Definition 14 of generic process pools. Indeed, the condition that the parameters $VAL, PID, CID, sender, receiver$ are the same for all instantiations of generic processes is assured by Definition 16. Recall from Section 7.3 that we assume all events, except for *trans*-events, to be implicitly tagged with the respective process ids. Hence, the only events that can be shared by two processes are *trans*-events.

Let us introduce a useful property that intuitively states that each command in a process correctly uses channel connections, i.e., sends and receives only on channels it is supposed to. This is a desirable property that we will from here on impose on DMWL process pools.

Definition 18 *A* command $C \in CMD$ and a process identifier *pid* comply with functions *sender*, *receiver if for every send command* $\mathsf{send}(cid, Exp)$*, which is a subcommand of* C*, holds* $sender(cid) = pid$ *and for every receive command* $\mathsf{receive}(cid, var)$ *or* $\mathsf{if\text{-}receive}(cid, var, C_1, C_2)$*, which is a subcommand of* C*, holds* $receiver(cid) = pid$*.*

7.5 Adequacy of DMWL Process Pools

In this section, we extend the results from Section 5 to distributed DMWL programs. In Theorem 6, we will show that every trace of a DMWL process pool models a behavior that complies with the semantics of DMWL. In Theorem 7, we will show that for every behavior that complies with the semantics of DMWL, there is a trace of the corresponding process pool that models this behavior. Lemma 7, 8, and 9 (extensions of Lemmas 1–3) are helpful for proving these theorems and also Theorems 8 and 9.

Notational Conventions. Throughout this subsection and Sections 7.6–7.7, we assume that *Pid* is a finite set of process identifiers, i.e., $Pid = \{pid_1, \ldots, pid_n\} \subseteq PID$, for $pid \in Pid$, $initthread^{pid} \in THREAD$, $SES^{pid} = (S^{pid}, s_0^{pid}, E^{pid}, I^{pid}, O^{pid}, T^{pid})$ is defined by $SES^{pid} = DMWLProcess(pid, initthread^{pid})$. $SES = (S, s_0, E, I, O, T)$ models the resulting DMWL process pool, i.e., is defined by $SES = ||_{pid \in Pid} SES^{pid}$. Moreover, recall from Section 7.3 that no process can send to itself, i.e., for all $cid \in CID$ holds $sender(cid) \neq receiver(cid)$. Recall from Section 7.4 that we assume all commands in a process pool to comply with *sender*, *receiver* and the respective *pid*.

We first define functions $\mathsf{channel}$ and config that, respectively, extract a channel status function or the configuration of a process from a state of the composed *SES*. Here and further, we use $\mathit{outbuf}(\mathit{cid})$ to denote the function that returns $\langle\rangle$ in case *outbuf* is empty or contains an entry with an identifier different from *cid*; and returns the second element of *outbuf* otherwise, i.e., $\mathit{outbuf}(\mathit{cid}) = \mathit{val}$ if $\mathit{outbuf} = (\mathit{cid}, \mathit{val})$.

Definition 19 *Let* $\mathit{pid}_i = \mathit{sender}(\mathit{cid})$ *and* $\mathit{pid}_r = \mathit{receiver}(\mathit{cid})$. *Moreover, let* $s|_{\mathit{pid}_i} = (\dots, \mathit{outbuf}_s^{\mathit{pid}_i}, \dots)$ *and* $s|_{\mathit{pid}_r} = (\dots, \mathit{inbuf}_s^{\mathit{pid}_r}, \mathit{pending}_s^{\mathit{pid}_r}, \dots)$. *The function* $\mathsf{channel} : S \to (\mathit{CID} \to \mathit{VAL}^*)$ *is defined by:*

$$\mathsf{channel}(s) : \mathit{cid} \mapsto \mathit{outbuf}_s^{\mathit{pid}_i}(\mathit{cid}).\, \mathit{inbuf}_s^{\mathit{pid}_r}(\mathit{cid}).\, \mathit{pending}_s^{\mathit{pid}_r}(\mathit{cid})$$

where, if pid_i *(or* pid_r*) is not in the process pool, then* $\mathit{outbuf}_s^{\mathit{pid}_i}(\mathit{cid}) = \langle\rangle$ *(or* $\mathit{inbuf}_s^{\mathit{pid}_r}(\mathit{cid}) = \langle\rangle$ *and* $\mathit{pending}_s^{\mathit{pid}_r}(\mathit{cid}) = \langle\rangle$*, respectively).*

Definition 20 $\mathit{config} : S \to \mathit{PID} \to [(\vec{\mathit{CMD}} \times (\mathit{VAR} \to \mathit{VAL})) \cup \{\bot\}]$ *is defined by:*

$$\mathit{config}(s, \mathit{pid}) = \begin{cases} \bot & \text{, if } \mathit{pid} \notin \mathit{Pid} \\ (\mathrm{cseq}(\mathit{thread}_s), \mathit{mem}_s), & \text{if } \mathit{pid} \in \mathit{Pid} \text{ and } s|_{\mathit{pid}} = (\mathit{mem}_s, \mathit{thread}_s \dots) \end{cases}$$

Lemma 7 *If* s *is a reachable state for the composed SES,* $\mathit{pid} \in \mathit{Pid}$ *is a process identifier, and* $s|_{\mathit{pid}} = (\dots, \mathit{thread}_s, \mathit{atid}_s, \dots, \mathit{executed}_s, \dots, \mathit{outbuf}_s)$ *then*

- $\mathit{executed}_s = \mathit{ff} \wedge \mathit{atid}_s \neq \bot \wedge \mathit{thread}_s(\mathit{atid}_s) \notin \{\bot, \top, \langle\rangle\} \wedge \mathit{outbuf}_s = \langle\rangle$,
- $\mathit{executed}_s = \mathit{tt} \wedge \mathit{atid}_s \neq \bot \wedge \mathit{outbuf}_s \neq \langle\rangle$,
- $\mathit{executed}_s = \mathit{tt} \wedge \mathit{atid}_s \neq \bot \wedge \mathit{outbuf}_s = \langle\rangle$, *or*
- $\mathit{executed}_s = \mathit{ff} \wedge \mathit{atid}_s = \bot \wedge \mathit{outbuf}_s = \langle\rangle$ *holds.*

Lemma 8 *Let* s, s' *be reachable states for the composed SES,* $\mathit{pid} \in \mathit{Pid}$, $e \in E_{\mathit{local}}^{\mathit{DMWL}} \cap E^{\mathit{pid}}$ *with* $s \xrightarrow{e} s'$, $s|_{\mathit{pid}} = (\mathit{mem}_s, \mathit{thread}_s, \mathit{atid}_s, \dots, \mathit{executed}_s, \dots, \mathit{pending}_s, \mathit{outbuf}_s)$, *and* $s'|_{\mathit{pid}} = (\mathit{mem}_{s'}, \mathit{thread}_{s'}, \mathit{atid}_{s'}, \dots, \mathit{outbuf}_{s'})$. *Assume* $\mathit{executed}_s = \mathit{ff}$, $\mathit{atid}_s \neq \bot$, *and if* $e = \mathit{ite\text{-}rcv}^{\mathit{ff}}(\mathit{cid}, \mathit{var}, \mathit{val}, C_1, C_2)$ *then* $\mathsf{channel}(s)(\mathit{cid}) = \langle\rangle$.

- *If* $e \neq \mathit{fork}(C, D_1 \dots D_n)$ *then*
  $\langle\!|\mathit{thread}_s(\mathit{atid}_s), \mathit{mem}_s, \mathsf{channel}(s)|\!\rangle \twoheadrightarrow \langle\!|\mathit{thread}_{s'}(\mathit{atid}_{s'}), \mathit{mem}_{s'}, \mathsf{channel}(s')|\!\rangle$.
- *If* $e = \mathit{fork}(C, D_1 \dots D_n)$ *then*
  $\langle\!|\mathit{thread}_s(\mathit{atid}_s), \mathit{mem}_s, \mathsf{channel}(s)|\!\rangle$
  $\twoheadrightarrow \langle\!|\mathit{thread}_{s'}(\mathit{atid}_s.0) \dots \mathit{thread}_{s'}(\mathit{atid}_s.n), \mathit{mem}_{s'}, \mathsf{channel}(s')|\!\rangle$.

The following theorem shows that every trace of a DMWL process pool, which is *closed*, models a behavior that complies with the semantics of DMWL. We say that a DMWL process pool is *closed* if all channels are connected, i.e., if for all $\mathit{cid} \in \mathit{CID}$ holds $\mathit{sender}(\mathit{cid}) \in \mathit{Pid} \iff \mathit{receiver}(\mathit{cid}) \in \mathit{Pid}$.

Theorem 6 *Let* s, s' *be reachable states for the composed SES and* $\gamma \in E^*$. *Assume that SES is closed. If* $s \stackrel{\gamma}{\Longrightarrow} s'$ *and* γ *contains no setvar-events then*

$$\begin{aligned} &\lhd \mathit{config}(s, \mathit{pid}_1), \dots, \mathit{config}(s, \mathit{pid}_n); \mathsf{channel}(s) \rhd \\ &\twoheadrightarrow^* \lhd \mathit{config}(s', \mathit{pid}_1), \dots, \mathit{config}(s', \mathit{pid}_n); \mathsf{channel}(s') \rhd \end{aligned}$$

Lemma 9 *Let $s \in S$ be a reachable state for the composed SES, $pid \in Pid$, $s|_{pid} = (mem_s, thread_s, atid_s, \ldots, executed_s, \ldots, pending_s, outbuf_s)$, $C' \in CMD$, $D_1 \ldots D_k \in \vec{CMD}$, $mem' : VAR \to VAL$, and $\sigma' : CID \to VAL^*$. Assume $executed_s = ff$, $atid_s \neq \bot$, and, moreover, if $first(thread_s(atid_s)) \in \{\mathsf{receive}(cid', \ldots), \mathsf{if\text{-}receive}(cid', \ldots)\}$ and $pending_s(cid') = \langle\rangle$ then $\mathsf{channel}(s)(cid') = \langle\rangle$.*

1. *If $\langle\!|thread_s(atid_s), mem_s, \mathsf{channel}(s)|\!\rangle \twoheadrightarrow \langle\!|C', mem', \sigma'|\!\rangle$ then $e \in E^{DMWL}_{local} \cap E^{pid}$ and $s' \in S$ exist with $s \xrightarrow{e} s'$, $s'|_{pid} = (mem_{s'}, thread_{s'}, atid_{s'}, \ldots, executed_{s'}, \ldots)$, $mem_{s'} = mem'$, $thread_{s'}(atid_s) = C'$, $atid_{s'} = atid_s$, $executed_{s'} = tt$, $\mathsf{channel}(s') = \sigma'$. Moreover, it holds $thread_{s'}(tid) = thread_s(tid)$ for $tid \neq atid_s$ and $s'|_{pid'} = s|_{pid'}$ for $pid' \in Pid$ with $pid' \neq pid$.*

2. *If $\langle\!|thread_s(atid_s), mem_s, \mathsf{channel}(s)|\!\rangle \twoheadrightarrow \langle\!|C' D_1 \ldots D_k, mem', \sigma'|\!\rangle$ (with $k \geq 1$) then an event $e \in E^{DMWL}_{local} \cap E^{pid}$ and $s' \in S$ exist with $s \xrightarrow{e} s'$, $s'|_{pid} = (mem_{s'}, thread_{s'}, atid_{s'}, \ldots, executed_{s'}, \ldots)$, $mem_{s'} = mem'$, $atid_{s'} = atid_s$, $executed_{s'} = tt$, and $\mathsf{channel}(s') = \sigma'$. Moreover, it holds $thread_{s'}(atid_s) = \top$, $thread_{s'}(atid_s.0) = C'$, $thread_{s'}(atid_s.i) = D_i$ for $i \in \{1, \ldots, k\}$, $thread_{s'}(tid) = thread_s(tid)$ for $tid \notin \{atid_s, atid_s.0, \ldots, atid_s.k\}$, and $s'|_{pid'} = s|_{pid'}$ for $pid' \in Pid$ with $pid' \neq pid$.*

The following theorem shows that for every behavior of a distributed DMWL program that is closed there is a trace of the corresponding DMWL process pool.

Theorem 7 *Let s be a reachable state for the composed SES. Assume that SES is closed. Moreover, assume for all $pid \in Pid$ holds $atid_s = \bot$ and $executed_s = ff$ where $s|_{pid} = (\ldots, atid_s, \ldots, executed_s, \ldots)$. Let $(\vec{C'_{pid_1}}, mem'_{pid_1}) \ldots (\vec{C'_{pid_n}}, mem'_{pid_n})$ be a sequence of pairs, each consisting of a command vector and a memory.*
If $\lhd\, \mathsf{config}(s, pid_1), \ldots, \mathsf{config}(s, pid_n); \mathsf{channel}(s) \rhd$
$\twoheadrightarrow^* \lhd(\vec{C'_{pid_1}}, mem'_{pid_1}), \ldots, (\vec{C'_{pid_n}}, mem'_{pid_n}); \sigma' \rhd$
then there exist $\gamma \in E^*$ and $s' \in S$ of the composed SES with $s \stackrel{\gamma}{\Longrightarrow} s'$, γ contains no setvar-events, $\mathsf{config}(s', pid) = (\vec{C'_{pid}}, mem'_{pid})$ for all $pid \in Pid$, and $\mathsf{channel}(s') = \sigma'$.

7.6 Soundness and Completeness Results

In this subsection, we present the soundness and completeness results for DMWL programs. We show that a DMWL program is secure iff the corresponding DMWL process satisfies *SecProp*. In distinction to Subsection 7.5, we assume to have a single process *pid*, so that $Pid = \{pid\}$. Otherwise, the same notational conventions apply. Note that the definition of the channel status extraction $\mathit{channel} : S \to (CID \to VAL^*)$ in the case of one process can be expressed as:

$$\mathsf{channel}(s) : cid \mapsto \begin{cases} inbuf_s^{pid}(cid).pending_s^{pid}(cid), & \text{if } receiver(cid) = pid \\ outbuf_s^{pid}(cid), & \text{if } sender(cid) = pid \end{cases}$$

The next auxiliary lemma is analogous to Lemma 4 from Section 6. It will be used in the proof of Theorem 8.

Lemma 10 *Assume a DMWL program C is secure. Suppose $\beta \in$ Tr is a trace for DMWLProcess(pid, C) such that $s_0 \stackrel{\beta}{\Longrightarrow} s$. Then $\forall tid.\, live(s, tid) \Longrightarrow thread_s(tid) \approxeq_L thread_s(tid)$.*

Let us conclude this subsection by stating the soundness and completeness results.

Theorem 8 (Soundness) *If a DMWL program C is secure then the corresponding process DMWLProcess(pid, C) satisfies the security property SecProp.*

Theorem 9 (Completeness) *A DMWL program C is secure whenever the corresponding process DMWLProcess(pid, C) satisfies the security property SecProp.*

The proof of the completeness theorem for DMWL makes use of both *BSI* and *BSD* security predicates from *SecProp*. Interestingly, *BSD* is not necessary in the completeness result for MWL (Theorem 5) [27]. Indeed, the deletion of a confidential *setvar*$(h, \cdot)$-event can be simulated by the insertion of a *setvar*$(h, \cdot)$-event that undoes the change to the memory of a thread pool. However, this technique does not apply to confidential *trans*-events occurring in DMWL process pools because inserting such events might not comply with the semantics of communicating thread pools (cf. the proof for further details).

7.7 Compositionality

In this section, we demonstrate that *SecProp* is preserved under composition of DMWL processes. This result holds even though generalized noninterference is, in general, not preserved under composition as demonstrated by McCullough [28]. Our technique for establishing compositionality is to develop a composable property *CSecProp* and prove that it is equivalent to *SecProp* for DMWL processes.[12] In Theorem 10, we demonstrate that *CSecProp*, indeed, is preserved under the composition of processes. In order to represent *CSecProp* in the assembly kit, we introduce the building block *forward correctable insertion* (abbreviated by *FCI*) [26]. Besides a view $\mathcal{V}$ and a set *Tr* of traces, this *BSP* takes two sets $\nabla, \Upsilon \subseteq E$ as parameters.

FCI demands that the insertion of a confidential event into a trace yields, again, a possible trace. In this aspect, *FCI* is related to *BSI* (cf. Section 2). Technically, $FCI_{\mathcal{V}}^{\nabla,\Upsilon}(Tr)$ demands that a confidential event $c \in C \cap \Upsilon$ can be inserted before the occurrence of a visible event $v \in V \cap \nabla$ if the sequence α that follows v contains no confidential input events from C. During the insertion of c, the trace may be adapted, however, adaptations are only allowed after the occurrence of v.[13] I.e., α may be

[12] *CSecProp* can be regarded as a weakened version of Johnson and Thayer's forward correctability [20].

[13] In [26] a more general definition of *FCI* is presented that takes another parameter Δ and permits adaptations in $N \cap \Delta$ before v. However, this flexibility is not needed here.

changed to α' where visible events in V and events in C must remain unchanged.

$$
\begin{aligned}
&\mathit{FCI}^{\nabla,\Upsilon}_{V,N,C}(\mathit{Tr})\\
&\equiv \forall \alpha,\beta \in E^*.\, \forall c \in C \cap \Upsilon.\, \forall v \in V \cap \nabla.\, ((\beta.\langle v\rangle.\alpha \in \mathit{Tr} \wedge \alpha|_C = \langle\rangle)\\
&\quad \Longrightarrow\ \exists \alpha' \in E^*.\, (\alpha'|_V = \alpha|_V \wedge \alpha'|_C = \langle\rangle \wedge \beta.\langle c.v\rangle.\alpha' \in \mathit{Tr}))
\end{aligned}
$$

The security predicate *CSP* (for "composable security predicate") is defined by:

$$\mathit{CSP}^{\nabla,\Upsilon}_{\mathcal{V}}(\mathit{Tr}) = \mathit{BSI}_{\mathcal{V}}(\mathit{Tr}) \wedge \mathit{BSD}_{\mathcal{V}}(\mathit{Tr}) \wedge \mathit{FCI}^{\nabla,\Upsilon}_{\mathcal{V}}(\mathit{Tr})$$

For the purposes of the current article, we choose ∇ and Υ wrt a set $\mathit{Pid} \subseteq \mathit{PID}$.

$$\nabla^{\mathit{Pid}} \quad = \quad \Upsilon^{\mathit{Pid}} \quad = \quad \{\mathit{trans}(\mathit{cid},\mathit{val}) \mid \mathit{receiver}(\mathit{cid}) \in \mathit{Pid}, \mathit{sender}(\mathit{cid}) \notin \mathit{Pid}\}\ .$$

Given the view $\mathcal{V}_{\mathit{pid}}$ of a thread pool with identifier *pid*, we refer to the security property $(\{\mathcal{V}_{\mathit{pid}}\}, \mathit{CSP}^{\nabla^{\{\mathit{pid}\}},\Upsilon^{\{\mathit{pid}\}}})$ as *CSecProp* (for "composable security property").

Below, we assume that for each $\mathit{pid} \in \mathit{Pid}$, $\mathit{ES}^{\mathit{pid}} = (E^{\mathit{pid}}, I^{\mathit{pid}}, O^{\mathit{pid}}, \mathit{Tr}^{\mathit{pid}})$ is the event system for $\mathit{SES}^{\mathit{pid}}$, i.e., $\mathit{ES}^{\mathit{pid}} = \mathit{ES}_{\mathit{SES}^{\mathit{pid}}}$, that $\mathcal{V}^{\mathit{pid}} = (V^{\mathit{pid}}, N^{\mathit{pid}}, C^{\mathit{pid}}) = \mathcal{V}_{TP_{\mathit{pid}}}$ is the view for process *pid*, that $\nabla^{\mathit{pid}} = \nabla^{\{\mathit{pid}\}} = \Upsilon^{\mathit{pid}}$, and that $\mathit{SecProp}^{\mathit{pid}} = (\{\mathcal{V}^{\mathit{pid}}\}, \mathit{BSI} \wedge \mathit{BSD})$ and $\mathit{CSecProp}^{\mathit{pid}} = (\{\mathcal{V}^{\mathit{pid}}\}, \mathit{CSP}^{\nabla^{\{\mathit{pid}\}},\Upsilon^{\{\mathit{pid}\}}})$ are the security properties for process *pid*. Moreover, we assume that $\mathit{SES} = (S, s_0, E, I, O, T)$ is the process pool resulting from the composition of these DMWL processes, i.e., $\mathit{SES} = \|_{\mathit{pid}\in\mathit{Pid}} \mathit{SES}^{\mathit{pid}}$, that $\mathit{ES} = (E, I, O, \mathit{Tr})$ is the corresponding event system, i.e., $\mathit{ES} = \mathit{ES}_{\mathit{SES}}$, that $\mathcal{V}^{\mathit{Pid}} = (V, N, C) = \mathcal{V}_{PP_{\mathit{Pid}}}$ is the view for the process pool, and that $\mathit{SecProp}^{\mathit{Pid}} = (\{\mathcal{V}^{\mathit{Pid}}\}, \mathit{BSI} \wedge \mathit{BSD})$ and $\mathit{CSecProp}^{\mathit{Pid}} = (\{\mathcal{V}^{\mathit{Pid}}\}, \mathit{CSP}^{\nabla^{\mathit{Pid}},\Upsilon^{\mathit{Pid}}})$ are the security properties for the process pool.

The following "zipping lemma" will be helpful for proving that *CSecProp* is preserved under composition of DMWL processes (cf. Theorem 10). The proof technique that we use is similar to the one used in [20] for the composability of forward correctability.[14] However, our security property is slightly weaker than forward correctability.[15]

Lemma 11 (Zipping Lemma) *Let $\tau \in E^*$, $\lambda \in V^*$, and $t_{\mathit{pid}} \in E^{\mathit{pid}*}$ for pid $\in$ Pid. If $\tau \in \mathit{Tr}$, $\tau|_{E^{\mathit{pid}}}.t_{\mathit{pid}} \in \mathit{Tr}^{\mathit{pid}}$, $t_{\mathit{pid}}|_V = \lambda|_{E^{\mathit{pid}}}$, and $\mathit{ES}^{\mathit{pid}}$ satisfies $\mathit{CSP}^{\nabla^{\mathit{pid}},\Upsilon^{\mathit{pid}}}_{\mathcal{V}^{\mathit{pid}}}(\mathit{Tr}^{\mathit{pid}})$ for $\mathit{pid} \in \mathit{Pid}$ then there is a sequence $t \in E^*$ with $\tau.t \in \mathit{Tr}$, $t|_V = \lambda$, and $t|_C = \langle\rangle$.*

The following theorem shows that *CSecProp* is, indeed, preserved under the composition of DMWL processes.

[14] For a more general account of this proof technique in the context of *MAKS*, we refer the interested reader to [26].

[15] Technically, this difference to forward correctability results from that we do not require all high input events to be forward correctable. In particular, high *setvar*-events are not contained in Υ although they are high input events. Moreover, events in $C \cap \Upsilon$ need not be forward correctable wrt all low-level input events, rather, only wrt *trans*-events on incoming low-level channels. Recall that forward correctability, as defined in [20], requires that all high input events are forward correctable wrt all low-level input events. Another difference is that we require only *FCI* but no corresponding *BSP* for forward correctable deletion.

Theorem 10 (Compositionality) *If $CSecProp^{pid}$ holds for each ES^{pid} then $CSecProp^{Pid}$ holds for ES, the composition of these DMWL processes.*

The following lemma implies that any DMWL process that satisfies *SecProp* also satisfies *CSecProp*.

Lemma 12 *If $BSI_{\mathcal{V}^{pid}}(Tr^{pid})$ holds for $ES_{SES^{pid}}$ then $FCI^{\nabla^{pid},\Upsilon^{pid}}_{\mathcal{V}^{pid}}(Tr^{pid})$ also holds.*

Theorem 10 and Lemma 12 give rise to the following corollary. This corollary ensures that *SecProp* is preserved under the composition of processes.

Corollary 1 *If for each $pid \in Pid$, ES^{pid} satisfies $SecProp^{pid}$ then the composed system ES satisfies $SecProp^{Pid}$.*

Recalling from Section 3.2 that *SecProp* corresponds to generalized noninterference [28], Corollary 1 can be reformulated by: *if each process in a process pool satisfies generalized noninterference then the overall process pool also satisfies generalized noninterference*. This result holds despite generalized noninterference is, in general, not preserved under composition [28] and no specialized form of composition is employed for which generalized noninterference is known to be preserved (like cascade [31, 47] or absence of communication cycles with less than three components [48]). Corollary 1 provides a basis for analyzing complex systems in a modular way. I.e., security of the overall system is derived by establishing local security for each process. Since the language-based techniques are also compositional (on the level of commands), establishing security of a process, again, can be reduced to establishing security of each command (also cf. the motivation for this approach in Section 7.2 and Figure 12).

Note that local security of each process is, in general, not necessary for global security. The above corollary only ensures that it is sufficient. Hence, there are process pools that satisfy *SecProp* although some processes in the pool do not satisfy *SecProp* individually. This overrestrictiveness of the local security condition seems to be a price that needs to be paid in order to allow for modular system development and for the application of efficient language-based techniques. One important advantage of our compositionality result in this respect (together with the rigorous relation of the language-based techniques and the trace-based properties) is that hybrid techniques can be used to verify the security of the overall system. I.e., one tries to verify local security individually for every process using the most efficient technique available, e.g., by performing security-type inference. For groups of processes for which the local security condition does not hold, the trace-based property can be verified directly. Global security of the overall distributed system follows from the compositionality of our security property together with the results in Sections 7.5 and 7.6.

8 Discussion and Future Work

Contributions. We have established a one-to-one correspondence between a time-sensitive definition of security for the multi-threaded programs of MWL (from [38]) and a security property based on traces of events that was originally developed in the context of a general security framework—the assembly kit *MAKS* (from [22, 23]). As a prerequisite for this, we had to model the semantics of MWL using state-event systems, which resulted in the specification of MWL thread pools. The development of this specification has been straightforward (although technically subtle). To us, it is appealing that generic thread pools, which served as an intermediate step in this process, are independent of MWL. We expect that this will allow the adaptation of other multi-threaded programming languages, e.g., Slam [19].

The main motivation of our work has been the objective to integrate the two kinds of security: the security of local computations and the security of their communications. Event-based security aims at protecting occurrences of events and programming-language-based security aims at protecting secret values. Our work is a step to aid in the systematic security analysis of complex (potentially distributed) systems where some of the components are (or shall be) implemented in a specific programming language. To the best of our knowledge this article is the first attempt to establish a rigorous connection between these two notions of security.

The connection suggests directions for mutual benefits where the two areas can borrow from each other (cf. Future Work). Already in this article, we have fruitfully exploited the rigorous relation between global and local language-based security in deriving a compositionality principle for DMWL, the multi-threaded language MWL augmented with message passing. We have shown that the global security of the overall system consisting of a collection of distributed DMWL programs is implied by the local security of each thread. This opens up the opportunity for reducing the security certification for the global system to a decentralized certification of the security of individual threads. This can be done by language-based techniques such as, e.g., security-type systems [45, 19, 42, 1, 44, 2, 38, 39, 35] or security verification [21]. For DMWL, this approach accommodates both local security and overall system's global security. This provides high security assurance without being too restrictive for the programmer. That interesting secure programs manipulating sensitive data can be written is illustrated by, e.g., efficient searching and sorting algorithms that comply to timing-sensitive security [3].

As a side effect, we have demonstrated how to use *MAKS* at the concrete example of the multi-threaded programming language MWL. Using the assembly kit has turned out to be very helpful in the identification of an appropriate security property. This application is also interesting because it shows how time-sensitive security can be specified in *MAKS*. For a different technique to address timing channels by explicit *tick*-events we refer to [13].

Bisimulation vs. Trace-based Equivalence. The reader familiar with transition-system-based semantics might be surprised by the fact that the article relates a bisimulation-based property of programs with a trace-based one. It is well-known that small-step bisimulation makes more distinctions than trace-based equivalence. It is also well-known that trace-based properties are usually not compositional whereas bisimulation-based ones often are. Nevertheless, we have been able to prove correctness and completeness results for our translation of the security property.

What made these developments possible in spite of the two major differences between the bisimulation-based and trace-based models? The crucial property is the deterministic nature of the transition system underlying strong low-bisimulation. Indeed, bisimulation is defined on deterministic transitions ensuring that two bisimilar thread pools have the same branching behavior.[16] This property is necessary for guaranteeing *scheduler-independent security*. Two programs have to have identical branching behavior in order to be indistinguishable for the attacker under a scheduler-independent low-bisimulation. Otherwise, the two programs in the then and else branches, respectively, of an if statement with a secret condition could be used to leak the secret condition through observing the branching behavior ([38] shows how to implement this attack using the properties of a particular scheduler).

Although the determinism of the transition system underlying bisimulation is the key feature to relating bisimulation-based and trace-based models, it is not crucial for the actual security definition of MWL[17]. For example, if MWL had a nondeterministic choice operator $[]$ then the nondeterministic program $l := 0 \mathbin{[]} l := 1$ would be considered secure under Definition 7. However, the two security definitions (Definition 7 and Definition 5 for MWL thread pools) would be no longer equivalent. Indeed, at no surprise, the completeness theorem (Theorem 5) would not hold. A counterexample is the program $\mathsf{if}\ h = 0\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$ where $C_1 = l := 0; (l := 1 \mathbin{[]} l := 2)$ and $C_2 = (l := 0; l := 1) \mathbin{[]} (l := 0; l := 2)$. This program is considered secure under the trace-based model (Definition 5) but not secure according to the bisimulation-based Definition 7. Note that, whether one intuitively considers this program as secure or not depends very much on the model of computation one has in mind. For a detailed investigation of this close relation between notions of information flow and models of computation (notions of equivalence) we refer to [34].

Future Work. Plans for future work are centered around further exploiting the connection between the two types of security that we have established in the present article. Promising directions include the adaptation of intransitive security policies[18] for MWL based on solutions that were proposed in the context of the assembly kit [24] and to progress towards a development method that allows for the stepwise development starting from abstract specifications and ending with concrete programs (cf. [25] for recent progress on the refinement of information flow properties). Another attractive direction of research is to apply the reduction techniques of [38] combined with the results of this article to reasoning about probabilistic security properties for event-based systems.

In this article, we have limited the consideration of communication primitives to three primitives: a nonblocking send together with a blocking and a nonblocking receive. In a separate study [36] we present a comparison of different communication primitives with respect to their impact on security. Besides the three above primitives, we have considered synchronous communication (represented by blocking send and receive) and channels for encrypted traffic where an attacker may observe the presence of

[16] Technically one can view the identifier *tid* of a thread that is chosen by the scheduler for the next transition as a label that is distinguished by the strong low-bisimulation. Hence, an occurrence of *schedule*(*tid*) reveals which branch has been chosen in the computation.

[17] Certain security properties can be defined through low determinism, as in [33, 32].

[18] Intransitive flow policies would provide a way to represent downgrading (and thus, e.g., secure encryption) in the multi-threaded while-language.

messages but not their contents. We have proposed a type system for the extension that enforces timing-sensitive security. This is a step toward realizing the compositionality principle for DMWL (cf. Contributions). Another interesting extension of the model would be to consider other I/O primitives in addition to *setvar*/*outvar* for interaction with the environment.

Only recently have there been attempts to address the problem of information flow for systems that are run on a combination of trusted and untrusted hosts. A notable example is the *secure partitioning* approach by Zdancewic et al. [50]. This approach allows for automatic partitioning of a sequential program with security annotations (that specify the levels of data confidentiality and host trust) into communicating programs that run on the available hosts and perform the original computation. However, there is no proof that the system enforces system-wide noninterference. Because mutual distrust and potential failure of the distributed components are intrinsic properties of many distributed systems, incorporating these properties in our model is another important goal for future work.

Acknowledgments

Thanks are due to Fabio Martinelli and David Sands for encouraging discussions. We are grateful to Geoffrey Smith and the anonymous reviewers for helpful comments. We also wish to thank Koen Claessen for useful feedback on bisimulation proofs. For useful comments on the presentation, we would like to thank Alexandra Heidger and Axel Schairer. The first author thanks Chalmers University of Technology for providing an inspiring working environment during his visit.

References

[1] M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A core calculus of dependency. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 147–160, Jan. 1999.

[2] J. Agat. Transforming out timing leaks. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 40–53, Jan. 2000.

[3] J. Agat and D. Sands. On confidentiality and algorithms. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 64–77, May 2001.

[4] G. R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, 2000.

[5] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall, 2nd edition, 1996.

[6] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[7] E. S. Cohen. Information transmission in computational systems. *ACM SIGOPS Operating Systems Review*, 11(5):133–139, 1977.

[8] E. S. Cohen. Information transmission in sequential programs. In R. A. DeMillo, D. P. Dobkin, A. K. Jones, and R. J. Lipton, editors, *Foundations of Secure Computation*, pages 297–335. Academic Press, 1978.

[9] B. Davey and H. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[10] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[11] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[12] R. Focardi and R. Gorrieri. A classification of security properties for process algebras. *Journal of Computer Security*, 3(1):5–33, 1995.

[13] R. Focardi, R. Gorrieri, and F. Martinelli. Information flow analysis in a discrete-time process algebra. In *Proceedings of the IEEE Computer Security Foundations Workshop*, pages 170–184, July 2000.

[14] S. N. Foley. A universal theory of information flow. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 116–122, Apr. 1987.

[15] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 11–20, Apr. 1982.

[16] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[17] J. W. Gray III. Toward a mathematical foundation for information flow security. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 21–35, May 1991.

[18] J. D. Guttman and M. E. Nadel. What needs securing? In *Proceedings of the IEEE Computer Security Foundations Workshop*, pages 34–57, June 1988.

[19] N. Heintze and J. G. Riecke. The SLam calculus: programming with secrecy and integrity. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 365–377, Jan. 1998.

[20] D. M. Johnson and F. J. Thayer. Security and the composition of machines. In *Proceedings of the IEEE Computer Security Foundations Workshop*, pages 72–89, June 1988.

[21] R. Joshi and K. R. M. Leino. A semantic approach to secure information flow. *Science of Computer Programming*, 37(1–3):113–138, 2000.

[22] H. Mantel. Possibilistic definitions of security – An assembly kit –. In *Proceedings of the IEEE Computer Security Foundations Workshop*, pages 185–199, July 2000.

[23] H. Mantel. Unwinding possibilistic security properties. In *Proceedings of the European Symposium on Research in Computer Security*, volume 1895 of *LNCS*, pages 238–254. Springer-Verlag, Oct. 2000.

[24] H. Mantel. Information flow control and applications – Bridging a gap –. In *Proceedings of the International Symposium of Formal Methods Europe*, volume 2021 of *LNCS*, pages 153–172. Springer-Verlag, Mar. 2001.

[25] H. Mantel. Preserving information flow properties under refinement. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 78–91, May 2001.

[26] H. Mantel. On the composition of secure systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 88–101, May 2002.

[27] H. Mantel and A. Sabelfeld. A generic approach to the security of multi-threaded programs. In *Proceedings of the IEEE Computer Security Foundations Workshop*, pages 126–142, June 2001.

[28] D. McCullough. Specifications for multi-level security and hook-up property. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 161–166, 1987.

[29] J. McLean. Security models and information flow. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 180–187, 1990.

[30] J. McLean. Proving noninterference and functional correctness using traces. *Journal of Computer Security*, 1(1):37–58, 1992.

[31] J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 79–93, May 1994.

[32] A. W. Roscoe. CSP and determinism in security modeling. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 114–127, May 1995.

[33] A. W. Roscoe, J. C. P. Woodcock, and L. Wulf. Non-interference through determinism. In *Proceedings of the European Symposium on Research in Computer Security*, volume 875 of *LNCS*, pages 33–53. Springer-Verlag, Nov. 1994.

[34] P. Ryan and S. Schneider. Process algebra and non-interference. In *Proceedings of the IEEE Computer Security Foundations Workshop*, pages 214–227, June 1999.

[35] A. Sabelfeld. The impact of synchronisation on secure information flow in concurrent programs. In *Proceedings of the Andrei Ershov International Conference on Perspectives of System Informatics*, volume 2244 of *LNCS*, pages 227–241. Springer-Verlag, July 2001.

[36] A. Sabelfeld and H. Mantel. Static confidentiality enforcement for distributed programs. In *Proceedings of the International Symposium on Static Analysis*, volume 2477 of *LNCS*, pages 376–394. Springer-Verlag, Sept. 2002.

[37] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communication*, 2002. To appear. Available via http://www.cs.cornell.edu/∼andrei.

[38] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proceedings of the IEEE Computer Security Foundations Workshop*, pages 200–214, July 2000.

[39] A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. *Higher Order and Symbolic Computation*, 14(1):59–91, Mar. 2001.

[40] G. Smith. A new type system for secure information flow. In *Proceedings of the IEEE Computer Security Foundations Workshop*, pages 115–125, June 2001.

[41] G. Smith. Weak probabilistic bisimulation for secure information flow. In *Proceedings of the Workshop on Issues in the Theory of Security*, Jan. 2002.

[42] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 355–364, Jan. 1998.

[43] D. Sutherland. A model of information. In *Proceedings of the 9th National Computer Security Conference*, pages 175–183, Sept. 1986.

[44] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *Journal of Computer Security*, 7(2–3):231–253, Nov. 1999.

[45] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[46] J. T. Wittbold and D. M. Johnson. Information flow in nondeterministic systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 144–161, May 1990.

[47] A. Zakinthinos and E. S. Lee. The composability of non-interference. In *Proceedings of the IEEE Computer Security Foundations Workshop*, pages 2–8, June 1995.

[48] A. Zakinthinos and E. S. Lee. How and why feedback composition fails. In *Proceedings of the IEEE Computer Security Foundations Workshop*, pages 95–101, June 1996.

[49] A. Zakinthinos and E. S. Lee. A general theory of security properties. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 94–102, May 1997.

[50] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *Proceedings of the ACM Symposium on Operating System Principles*, pages 1–14, Oct. 2001.

Appendix

Proof. [of Theorem 1] Assume that $\mathit{SES}^j = (S^j, s_0^j, E^j, I^j, O^j, T^j)$ for $j \in J$ and $||_{j \in J}\, \mathit{SES}^j = (S, s_0, E, I, O, T)$.

$$\begin{aligned}
\tau \in \mathit{Tr}_{||_{j\in J}\,\mathit{SES}^j} &\iff \exists s \in S.\, (s_0 \overset{\tau}{\Longrightarrow}_T s) \\
&\iff \exists s \in S. \forall j \in J.\, (s_0|_j \overset{\tau|_{E^j}}{\Longrightarrow}_{T^j} s|_j) \\
&\iff \forall j \in J.\, (\tau|_{E^j} \in \mathit{Tr}_{\mathit{SES}^j})
\end{aligned}$$

The second equivalence is established by an induction over τ. □

Proof. [of Lemma 1] There exists an event sequence $\gamma \in E^*$ such that $s_0 \overset{\gamma}{\Longrightarrow} s$ because s is reachable. The proof proceeds by induction on γ.

Base case ($\gamma = \langle\rangle$): $\mathit{executed}_{s_0} = \mathit{ff}$ and $\mathit{atid}_{s_0} = \bot$ according to Figure 2.

Step case ($\gamma = \delta.\langle e\rangle$): There exists $s_i \in S$ with $s_0 \overset{\delta}{\Longrightarrow} s_i$ and $s_i \overset{e}{\longrightarrow} s$. The induction hypothesis ensures that the proposition holds in s_i. We have to show that it also holds in s. If e is a *setvar*- or *outvar*-event then the proposition follows directly because these events do not affect the values of *executed*, *atid*, and *thread*. In the rest of the proof, we assume that e is no *setvar*- or *outvar*-event. According to the induction hypothesis, we have to distinguish three cases.

- Assume $\mathit{executed}_{s_i} = \mathit{ff} \wedge \mathit{atid}_{s_i} \neq \bot \wedge \mathit{thread}_{s_i}(\mathit{atid}_{s_i}) \notin \{\bot, \top, \langle\rangle\}$. The preconditions of events (cf. Section 3 and 4) imply $e \in E^{\text{MWL}}_{\mathit{local}}$. The postcondition of any $e \in E^{\text{MWL}}_{\mathit{local}}$ ensures $\mathit{executed}_s = \mathit{tt}$ and $\mathit{atid}_s \neq \bot$.
- $\mathit{executed}_{s_i} = \mathit{tt} \wedge \mathit{atid}_{s_i} \neq \bot$ implies $e = \mathit{yield}(\mathit{info})$ for some $\mathit{info} \in \mathit{INFO}$. The postcondition of *yield* ensures $\mathit{executed}_s = \mathit{ff}$ and $\mathit{atid}_s = \bot$.
- $\mathit{executed}_{s_i} = \mathit{ff} \wedge \mathit{atid}_{s_i} = \bot$ implies $e = \mathit{schedule}(\mathit{tid})$ for some $\mathit{tid} \in \mathit{TID}$. The postcondition ensures $\mathit{executed}_s = \mathit{ff}$, $\mathit{atid}_s \neq \bot$, $\mathit{thread}_s(\mathit{atid}_s) \notin \{\bot, \top, \langle\rangle\}$. □

Proof. [of Lemma 2] We make a case distinction on the event e.

skip Pre- and postcondition of *skip* imply $\mathit{first}(\mathit{thread}_s(\mathit{atid}_s)) = \mathsf{skip}$, $\mathit{mem}_{s'} = \mathit{mem}_s$, and $\mathit{thread}_{s'}(\mathit{atid}_{s'}) = \mathit{rest}(\mathit{thread}_s(\mathit{atid}_s))$. Rules Skip and Seq$_1$ in Figure 5 ensure $\langle\!|\mathit{thread}_s(\mathit{atid}_s), \mathit{mem}_s|\!\rangle \twoheadrightarrow \langle\!|\mathit{thread}_{s'}(\mathit{atid}_{s'}), \mathit{mem}_{s'}|\!\rangle$.

assign(var,val) We have $\mathit{first}(\mathit{thread}_s(\mathit{atid}_s)) = \mathit{var} := \mathit{Exp}$ for some expression Exp with $\mathit{Exp} \downarrow^{\mathit{mem}_s} \mathit{val}$, $\mathit{mem}_{s'} = \mathit{mem}_s[\mathit{var} \mapsto \mathit{val}]$, and $\mathit{thread}_{s'}(\mathit{atid}_{s'}) = \mathit{rest}(\mathit{thread}_s(\mathit{atid}_s))$. Rules Assign and Seq$_1$ ensure $\langle\!|\mathit{thread}_s(\mathit{atid}_s), \mathit{mem}_s|\!\rangle \twoheadrightarrow \langle\!|\mathit{thread}_{s'}(\mathit{atid}_{s'}), \mathit{mem}_{s'}|\!\rangle$.

$\boldsymbol{\mathit{ite}^{\mathit{tt}}}(B, C_1, C_2)$ We have $\mathit{first}(\mathit{thread}_s(\mathit{atid}_s)) = \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$, $B \downarrow^{\mathit{mem}_s} \mathit{tt}$, $\mathit{mem}_{s'} = \mathit{mem}_s$, and $\mathit{thread}_{s'}(\mathit{atid}_{s'}) = C_1; \mathit{rest}(\mathit{thread}_s(\mathit{atid}_s))$. Rules If$_{\mathit{tt}}$ and Seq$_1$ ensure $\langle\!|\mathit{thread}_s(\mathit{atid}_s), \mathit{mem}_s|\!\rangle \twoheadrightarrow \langle\!|\mathit{thread}_{s'}(\mathit{atid}_{s'}), \mathit{mem}_{s'}|\!\rangle$.

$\boldsymbol{\mathit{ite}^{\mathit{ff}}}(B, C_1, C_2)$ We have $\mathit{first}(\mathit{thread}_s(\mathit{atid}_s)) = \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$, $B \downarrow^{\mathit{mem}_s} \mathit{ff}$, $\mathit{mem}_{s'} = \mathit{mem}_s$, and $\mathit{thread}_{s'}(\mathit{atid}_{s'}) = C_2; \mathit{rest}(\mathit{thread}_s(\mathit{atid}_s))$. Rules If$_{\mathit{ff}}$ and Seq$_1$ ensure $\langle\!|\mathit{thread}_s(\mathit{atid}_s), \mathit{mem}_s|\!\rangle \twoheadrightarrow \langle\!|\mathit{thread}_{s'}(\mathit{atid}_{s'}), \mathit{mem}_{s'}|\!\rangle$.

while$^{\boldsymbol{tt}}(B, C_1)$ We have $\mathit{first}(\mathit{thread}_s(\mathit{atid}_s)) = \mathsf{while}\ B\ \mathsf{do}\ C_1$, $B \downarrow^{\mathit{mem}_s} \mathit{tt}$, $\mathit{mem}_{s'} = \mathit{mem}_s$, and $\mathit{thread}_{s'}(\mathit{atid}_{s'}) = C_1; \mathsf{while}\ B\ \mathsf{do}\ C_1; \mathit{rest}(\mathit{thread}_s(\mathit{atid}_s))$. Rules While_{tt} and Seq_1 ensure $\langle\!|\mathit{thread}_s(\mathit{atid}_s), \mathit{mem}_s|\!\rangle \twoheadrightarrow \langle\!|\mathit{thread}_{s'}(\mathit{atid}_{s'}), \mathit{mem}_{s'}|\!\rangle$.

while$^{\boldsymbol{ff}}(B, C_1)$ We have $\mathit{first}(\mathit{thread}_s(\mathit{atid}_s)) = \mathsf{while}\ B\ \mathsf{do}\ C_1$, $B \downarrow^{\mathit{mem}_s} \mathit{ff}$, $\mathit{mem}_{s'} = \mathit{mem}_s$, and $\mathit{thread}_{s'}(\mathit{atid}_{s'}) = \mathit{rest}(\mathit{thread}_s(\mathit{atid}_s))$. Rules While_{ff} and Seq_1 ensure $\langle\!|\mathit{thread}_s(\mathit{atid}_s), \mathit{mem}_s|\!\rangle \twoheadrightarrow \langle\!|\mathit{thread}_{s'}(\mathit{atid}_{s'}), \mathit{mem}_{s'}|\!\rangle$.

fork$(C, \vec{D})$ We have $\mathit{first}(\mathit{thread}_s(\mathit{atid}_s)) = \mathsf{fork}(CD_1 \ldots D_n)$, $\mathit{mem}_{s'} = \mathit{mem}_s$, $\mathit{thread}_{s'}(\mathit{atid}_s) = \top$, and $\mathit{thread}_{s'}(\mathit{atid}.0) = C; \mathit{rest}(\mathit{thread}(\mathit{atid}))$. For all $i \in \{1, \ldots, n\}$ holds $\mathit{thread}_{s'}(\mathit{atid}.i) = D_i$. Rules Fork and Seq_2 ensure $\langle\!|\mathit{thread}_s(\mathit{atid}_s), \mathit{mem}_s|\!\rangle \twoheadrightarrow \langle\!|\mathit{thread}_{s'}(\mathit{atid}_{s'}.0) \ldots \mathit{thread}_{s'}(\mathit{atid}_{s'}.n), \mathit{mem}_{s'}|\!\rangle$. □

Proof. [of Theorem 2] The proof proceeds by induction on the length of γ.

Base case ($\gamma = \langle\rangle$): The proposition holds because $s' = s$ and $\rightarrow^*$ is reflexive.

Step case ($\gamma = \langle e \rangle.\delta$): There exists $s_i \in S$ with $s \xrightarrow{e} s_i$ and $s_i \xRightarrow{\delta} s'$. If e is an *outvar*-event then $s_i = s$ and the proposition follows from the induction hypothesis. Assume that e is no *outvar*-event. We make a case distinction according to Lemma 1.

- Assume $\mathit{executed}_s = \mathit{ff} \wedge \mathit{atid}_s \neq \bot \wedge \mathit{thread}_s(\mathit{atid}_s) \notin \{\bot, \top, \langle\rangle\}$. The proposition follows from $e \in E^{\text{MWL}}_{\mathit{local}}$, Lemma 2, the frame-axioms for *thread* for events in $E^{\text{MWL}}_{\mathit{local}}$, Definition 9, rule Pick, and the induction hypothesis.
- Assume $\mathit{executed}_s = \mathit{tt} \wedge \mathit{atid}_s \neq \bot$. $e = \mathit{yield}(\mathit{ainfo}_s)$ holds. $\mathit{mem}_{s_i} = \mathit{mem}_s$ and $\mathit{cseq}(\mathit{thread}_{s_i}) = \vec{D}_s$ are implied by the postcondition of *yield*. The proposition follows from the induction hypothesis.
- Assume $\mathit{executed}_s = \mathit{ff} \wedge \mathit{atid}_s = \bot$. $e = \mathit{schedule}(\mathit{tid})$ for some $\mathit{tid} \in \mathit{TID}$. $\mathit{mem}_{s_i} = \mathit{mem}_s$ and $\mathit{cseq}(\mathit{thread}_{s_i}) = \vec{D}_s$ are implied by the postcondition of *schedule*. The proposition follows from the induction hypothesis. □

Proof. [of Lemma 3] Let $\mathcal{D}$ be a derivation of $\langle\!|\mathit{thread}_s(\mathit{atid}_s), \mathit{mem}_s|\!\rangle \twoheadrightarrow \langle\!|C', \mathit{mem}'|\!\rangle$ (or $\langle\!|\mathit{thread}_s(\mathit{atid}_s), \mathit{mem}_s|\!\rangle \twoheadrightarrow \langle\!|C' D_1 \ldots D_n, \mathit{mem}'|\!\rangle$). According to the operational semantics of MWL (cf. Figure 5) there must be exactly one application of one of the rules Skip, Assign, If_{tt}, If_{ff}, While_{tt}, While_{ff}, or Fork in $\mathcal{D}$.

We make a case distinction depending on which of these rules occurs in $\mathcal{D}$:

Skip and Seq_1 ensure $\mathit{first}(\mathit{thread}_s(\mathit{atid}_s)) = \mathsf{skip}$, $C' = \mathit{rest}(\mathit{thread}_s(\mathit{atid}_s))$, $\mathit{mem}' = \mathit{mem}_s$. For $e = \mathit{skip}$ there exists $s' \in S$ with $s \xrightarrow{e} s'$, $\mathit{thread}_{s'}(\mathit{atid}_{s'}) = \mathit{rest}(\mathit{thread}_s(\mathit{atid}_s))$, $\mathit{mem}_{s'} = \mathit{mem}_s$, $\mathit{atid}_{s'} = \mathit{atid}_s$, and $\mathit{executed}_{s'} = \mathit{tt}$. Moreover, for all *tid* with $\mathit{tid} \neq \mathit{atid}_s$ holds $\mathit{thread}_{s'}(\mathit{tid}) = \mathit{thread}_s(\mathit{tid})$.

Assign and Seq_1 ensure $\mathit{first}(\mathit{thread}_s(\mathit{atid}_s)) = \mathit{var} := \mathit{Exp}$ for some $\mathit{var} \in \mathit{VAR}$, $\mathit{val} \in \mathit{VAL}$, $\mathit{Exp} \in \mathit{EXP}$ with $\mathit{Exp} \downarrow^{\mathit{mem}_s} \mathit{val}$, $C' = \mathit{rest}(\mathit{thread}_s(\mathit{atid}_s))$, and $\mathit{mem}' = \mathit{mem}_s[\mathit{var} \mapsto \mathit{val}]$. For $e = \mathit{assign}(\mathit{var}, \mathit{val})$ there exists $s' \in S$ with $s \xrightarrow{e} s'$, $\mathit{thread}_{s'}(\mathit{atid}_{s'}) = \mathit{rest}(\mathit{thread}_s(\mathit{atid}_s))$, $\mathit{mem}_{s'} = \mathit{mem}_s[\mathit{var} \mapsto \mathit{val}]$, $\mathit{atid}_{s'} = \mathit{atid}_s$, and $\mathit{executed}_{s'} = \mathit{tt}$. Moreover, for all *tid* with $\mathit{tid} \neq \mathit{atid}_s$ holds $\mathit{thread}_{s'}(\mathit{tid}) = \mathit{thread}_s(\mathit{tid})$.

If$_{tt}$ and Seq$_1$ ensure $\mathit{first}(\mathit{thread}_s(\mathit{atid}_s)) = \text{if } B \text{ then } C_1 \text{ else } C_2$ for $C_1, C_2 \in \mathit{CMD}$, $B \in \mathit{BOOL}$ with $B \downarrow^{\mathit{mem}_s} \mathit{tt}$, $C' = C_1; \mathit{rest}(\mathit{thread}_s(\mathit{atid}_s))$, $\mathit{mem}' = \mathit{mem}_s$. For $e = \mathit{ite}^{\mathit{tt}}(B, C_1, C_2)$ there exists $s' \in S$ with $s \xrightarrow{e} s'$, $\mathit{thread}_{s'}(\mathit{atid}_{s'}) = C_1; \mathit{rest}(\mathit{thread}_s(\mathit{atid}_s))$, $\mathit{mem}_{s'} = \mathit{mem}_s$, $\mathit{atid}_{s'} = \mathit{atid}_s$, and $\mathit{executed}_{s'} = \mathit{tt}$. Moreover, for all tid with $\mathit{tid} \neq \mathit{atid}_s$ holds $\mathit{thread}_{s'}(\mathit{tid}) = \mathit{thread}_s(\mathit{tid})$.

If$_{\mathit{ff}}$ and Seq$_1$ ensure $\mathit{first}(\mathit{thread}_s(\mathit{atid}_s)) = \text{if } B \text{ then } C_1 \text{ else } C_2$ for $C_1, C_2 \in \mathit{CMD}$, $B \in \mathit{BOOL}$ with $B \downarrow^{\mathit{mem}_s} \mathit{ff}$, $C' = C_2; \mathit{rest}(\mathit{thread}_s(\mathit{atid}_s))$, $\mathit{mem}' = \mathit{mem}_s$. For $e = \mathit{ite}^{\mathit{ff}}(B, C_1, C_2)$ there exists $s' \in S$ with $s \xrightarrow{e} s'$, $\mathit{thread}_{s'}(\mathit{atid}_{s'}) = C_2; \mathit{rest}(\mathit{thread}_s(\mathit{atid}_s))$, $\mathit{mem}_{s'} = \mathit{mem}_s$, $\mathit{atid}_{s'} = \mathit{atid}_s$, and $\mathit{executed}_{s'} = \mathit{tt}$. Moreover, for all tid with $\mathit{tid} \neq \mathit{atid}_s$ holds $\mathit{thread}_{s'}(\mathit{tid}) = \mathit{thread}_s(\mathit{tid})$.

While$_{tt}$ and Seq$_1$ ensure $\mathit{first}(\mathit{thread}_s(\mathit{atid}_s)) = \text{while } B \text{ do } C_1$ for $C_1 \in \mathit{CMD}$, $B \in \mathit{BOOL}$ with $B \downarrow^{\mathit{mem}_s} \mathit{tt}$, $C' = C_1; \text{while } B \text{ do } C_1; \mathit{rest}(\mathit{thread}_s(\mathit{atid}_s))$, $\mathit{mem}' = \mathit{mem}_s$. For $e = \mathit{while}^{\mathit{tt}}(B, C_1)$ there exists $s' \in S$ with $s \xrightarrow{e} s'$, $\mathit{thread}_{s'}(\mathit{atid}_{s'}) = C_1; \text{while } B \text{ do } C_1; \mathit{rest}(\mathit{thread}_s(\mathit{atid}_s))$, $\mathit{mem}_{s'} = \mathit{mem}_s$, $\mathit{atid}_{s'} = \mathit{atid}_s$, and $\mathit{executed}_{s'} = \mathit{tt}$. For all tid with $\mathit{tid} \neq \mathit{atid}_s$ holds $\mathit{thread}_{s'}(\mathit{tid}) = \mathit{thread}_s(\mathit{tid})$.

While$_{\mathit{ff}}$ and Seq$_1$ ensure $\mathit{first}(\mathit{thread}_s(\mathit{atid}_s)) = \text{while } B \text{ do } C_1$ for $C_1 \in \mathit{CMD}$, $B \in \mathit{BOOL}$ with $B \downarrow^{\mathit{mem}_s} \mathit{ff}$, $C' = \mathit{rest}(\mathit{thread}_s(\mathit{atid}_s))$, $\mathit{mem}' = \mathit{mem}_s$. For $e = \mathit{while}^{\mathit{ff}}(B, C_1)$ there exists $s' \in S$ with $s \xrightarrow{e} s'$, $\mathit{thread}_{s'}(\mathit{atid}_{s'}) = \mathit{rest}(\mathit{thread}_s(\mathit{atid}_s))$, $\mathit{mem}_{s'} = \mathit{mem}_s$, $\mathit{atid}_{s'} = \mathit{atid}_s$, and $\mathit{executed}_{s'} = \mathit{tt}$. Moreover, for all tid with $\mathit{tid} \neq \mathit{atid}_s$ holds $\mathit{thread}_{s'}(\mathit{tid}) = \mathit{thread}_s(\mathit{tid})$.

Fork and Seq$_2$ ensure $\mathit{first}(\mathit{thread}_s(\mathit{atid}_s)) = \text{fork}(C D_1 \ldots D_k)$ for some $C \in \mathit{CMD}$ and $D_1 \ldots D_k \in \vec{\mathit{CMD}}$, $C'' = C; \mathit{rest}(\mathit{thread}_s(\mathit{atid}_s))$, and $\mathit{mem}' = \mathit{mem}_s$. For $e = \mathit{fork}(C, D_1 \ldots D_k)$ there exists $s' \in S$ with $s \xrightarrow{e} s'$, $\mathit{thread}_{s'}(\mathit{atid}_s) = \top$, $\mathit{thread}_{s'}(\mathit{atid}_s.0) = C''$, for all $i \in \{1, \ldots, k\}$ holds $\mathit{thread}_{s'}(\mathit{atid}_s.i) = D_i$, $\mathit{mem}_{s'} = \mathit{mem}_s$, $\mathit{atid}_{s'} = \mathit{atid}_s$, and $\mathit{executed}_{s'} = \mathit{tt}$. Moreover, for all tid with $\mathit{tid} \notin \{\mathit{atid}_s, \mathit{atid}_s.0, \ldots, \mathit{atid}_s.k\}$ holds $\mathit{thread}_{s'}(\mathit{tid}) = \mathit{thread}_s(\mathit{tid})$. □

Proof. [of Theorem 3] Assume a derivation $\mathcal{D}$ for $\langle\!|\vec{D}_s, \mathit{mem}_s|\!\rangle \rightarrow^* \langle\!|\vec{D'}, \mathit{mem}'|\!\rangle$. The proof proceeds by induction on n, the number of applications of the rule Pick in $\mathcal{D}$.

Base case ($n = 0$): $\mathcal{D}$ contains no rule applications at all. Hence, $\vec{D'} = \vec{D}_s$ and $\mathit{mem}' = \mathit{mem}_s$. Consequently, the proposition holds for $\gamma = \langle\rangle$ and $s' = s$.

Step case ($n = n' + 1$): There exists $\langle\!|\vec{D}_i, \mathit{mem}_i|\!\rangle$ with $\langle\!|\vec{D}_s, \mathit{mem}_s|\!\rangle \rightarrow \langle\!|\vec{D}_i, \mathit{mem}_i|\!\rangle$ and $\langle\!|\vec{D}_i, \mathit{mem}_i|\!\rangle \rightarrow^* \langle\!|\vec{D'}, \mathit{mem}'|\!\rangle$. Let $\mathcal{D}_i$ be a derivation of $\langle\!|\vec{D}_s, \mathit{mem}_s|\!\rangle \rightarrow \langle\!|\vec{D}_i, \mathit{mem}_i|\!\rangle$ and $\mathcal{D}'$ be a derivation of $\langle\!|\vec{D}_i, \mathit{mem}_i|\!\rangle \rightarrow^* \langle\!|\vec{D'}, \mathit{mem}'|\!\rangle$ with n' application of rule Pick. If there exists $\gamma_i \in E^*$ that contains no *setvar*-events and a state s_i with $s \overset{\gamma_i}{\Longrightarrow} s_i$, $\mathit{mem}_{s_i} = \mathit{mem}_i$, $\mathit{cseq}(\mathit{thread}_{s_i}) = \vec{D}_i$, $\mathit{atid}_{s_i} = \bot$, and $\mathit{executed}_{s_i} = \mathit{ff}$ then the proposition follows from the induction hypothesis.

Pick is applied once in $\mathcal{D}_i$. Let tid be the identifier of the thread that is selected in this application. $e_1 = \mathit{schedule}(\mathit{tid})$ is enabled in s. For $s_1 \in S$ with $s \xrightarrow{e_1} s_1$ holds $\mathit{thread}_{s_1} = \mathit{thread}_s$, $\mathit{mem}_{s_1} = \mathit{mem}_s$, $\mathit{atid}_{s_1} \neq \bot$, and $\mathit{executed}_{s_1} = \mathit{ff}$. According to the definition of rule Pick and Lemma 3, there exist $e_2 \in E^{\text{MWL}}_{\mathit{local}}$, $s_2 \in S$ with $s_1 \xrightarrow{e_2} s_2$, $\mathit{cseq}(\mathit{thread}_{s_2}) = \vec{D}_i$, $\mathit{mem}_{s_2} = \mathit{mem}_i$, $\mathit{atid}_{s_2} \neq \bot$, and $\mathit{executed}_{s_2} = \mathit{tt}$. $e_3 = \mathit{yield}(\mathit{ainfo}_{s_2})$ is enabled in s_2. There exists $s_i \in S$ with $s_2 \xrightarrow{e_3} s_i$. Thus, for

$\gamma_i = \langle e_1.e_2.e_3\rangle$ holds $s \stackrel{\gamma_i}{\Longrightarrow} s_i$, $\mathit{cseq}(\mathit{thread}_{s_i}) = \vec{D}_i$, $\mathit{mem}_{s_i} = \mathit{mem}_i$, $\mathit{atid}_{s_i} = \bot$, and $\mathit{executed}_{s_i} = \mathit{ff}$. □

Proof. [of Lemma 4] The proof is by induction on the length of β. In the base case $\beta = \langle\rangle$ we have $\forall \mathit{tid} \neq 0.\, \mathit{live}(s, \mathit{tid}) = \mathit{ff}$ and $\mathit{thread}_s(0) = \mathit{thread}_s(0) = C$. We have $C \approxeq_L C$ by the security of C.

By the inductive step, $s_0 \stackrel{\delta}{\Longrightarrow} t \stackrel{e}{\longrightarrow} s$ such that $\forall \mathit{tid}.\, \mathit{live}(t, \mathit{tid}) \Longrightarrow \mathit{thread}_t(\mathit{tid}) \approxeq_L \mathit{thread}_t(\mathit{tid})$. If e is not local, then no threads may be created or updated. Thus, e cannot affect thread_t so that $\mathit{thread}_t = \mathit{thread}_s$ which completes the inductive step for this case. Now suppose e is local. We need to show $\forall \mathit{tid}.\, \mathit{live}(s, \mathit{tid}) \Longrightarrow \mathit{thread}_s(\mathit{tid}) \approxeq_L \mathit{thread}_s(\mathit{tid})$. We will appeal to Lemma 2 in order to match the transition in the semantics of MWL and unwind Definition 6 of strong low-bisimulation. We have two cases on e:

$\boldsymbol{\mathit{first}(\mathit{thread}_t(\mathit{atid}_t)) \neq \textsf{fork}(\cdot\cdot)}$ We have that $\mathit{executed}_t = \mathit{ff}$, $\mathit{atid}_t \neq \bot$, $\mathit{live}(t, \mathit{atid}_t)$, and $t \stackrel{e}{\longrightarrow} s$ hold. By applying Lemma 2 we obtain $\langle\!| \mathit{thread}_t(\mathit{atid}_t), \mathit{mem}_t |\!\rangle \twoheadrightarrow \langle\!| \mathit{thread}_s(\mathit{atid}_s), \mathit{mem}_s |\!\rangle$. By the induction hypothesis, we deduce $\mathit{thread}_t(\mathit{atid}_t) \approxeq_L \mathit{thread}_t(\mathit{atid}_t)$. Unwinding Definition 6 of strong low-bisimulation, there exist C' and mem' such that $\langle\!| \mathit{thread}_t(\mathit{atid}_t), \mathit{mem}_t |\!\rangle \twoheadrightarrow \langle\!| C', \mathit{mem}' |\!\rangle$ so that both $\mathit{thread}_s(\mathit{atid}_s) \approxeq_L C'$ and $\mathit{mem}_s =_L \mathit{mem}'$ hold. Due to the fact that $\twoheadrightarrow$-transitions are deterministic, it must be the case that $C' = \mathit{thread}_s(\mathit{atid}_s)$ and $\mathit{mem}' = \mathit{mem}_s$. This gives $\mathit{thread}_s(\mathit{atid}_s) \approxeq_L \mathit{thread}_s(\mathit{atid}_s)$. Because no other tid's are affected, the proof for this case is completed.

$\boldsymbol{\mathit{first}(\mathit{thread}_t(\mathit{atid}_t)) = \textsf{fork}(\cdot\cdot)}$ We have that $\mathit{executed}_t = \mathit{ff}$, $\mathit{atid}_t \neq \bot$, $\mathit{live}(t, \mathit{atid}_t)$, and $t \stackrel{e}{\longrightarrow} s$ hold. By applying Lemma 2 we obtain $\langle\!| \mathit{thread}_t(\mathit{atid}_t), \mathit{mem}_t |\!\rangle \twoheadrightarrow \langle\!| \mathit{thread}_s(\mathit{atid}_t.0) \ldots \mathit{thread}_s(\mathit{atid}_t.n), \mathit{mem}_s |\!\rangle$ where $n \in \mathbb{N}$ is the maximal natural number for which $\mathit{thread}_s(\mathit{atid}_t.n) \neq \bot$ holds. According to the induction hypothesis, $\mathit{thread}_t(\mathit{atid}_t) \approxeq_L \mathit{thread}_t(\mathit{atid}_t)$. By Definition 6 of strong low-bisimulation, there exist a command C', a sequence $\vec{D} = \langle D_1 \ldots D_n\rangle$, and mem' such that $\langle\!| \mathit{thread}_t(\mathit{atid}_t), \mathit{mem}_t |\!\rangle \twoheadrightarrow \langle\!| C'\vec{D}, \mathit{mem}' |\!\rangle$ where $\mathit{thread}_s(\mathit{atid}_t.0) \approxeq_L C'$, $\mathit{thread}_s(\mathit{atid}_t.1) \approxeq_L D_1, \ldots, \mathit{thread}_s(\mathit{atid}_t.n) \approxeq_L D_n$ and $\mathit{mem}_s =_L \mathit{mem}'$. Since $\twoheadrightarrow$-transitions are deterministic, we receive $C' = \mathit{thread}_s(\mathit{atid}_t.0), D_1 = \mathit{thread}_s(\mathit{atid}_t.1), \ldots, D_n = \mathit{thread}_s(\mathit{atid}_t.n)$ and $\mathit{mem}' = \mathit{mem}_s$. We receive $\mathit{thread}_s(\mathit{atid}_t.i) \approxeq_L \mathit{thread}_s(\mathit{atid}_t.i)$ for $i \in \{0, \ldots, n\}$. Because $\mathit{live}(s, \mathit{atid}_t) = \mathit{ff}$ and that no other tid's have been affected, the proof is completed. □

Proof. [of Theorem 4] Assume that C is secure. We need to show that $\mathit{MWLPool}(C)$ satisfies *SecProp*. According to Section 3.2, to satisfy *SecProp* we need to prove $\mathit{BSI}_{\mathcal{V}_{TP}}(\mathit{Tr})$ and $\mathit{BSD}_{\mathcal{V}_{TP}}(\mathit{Tr})$ where $\mathit{Tr} = \mathit{Tr}_{\mathit{MWLPool}(C)}$. Let us prove $\mathit{BSI}_{\mathcal{V}_{TP}}(\mathit{Tr})$. The proof for $\mathit{BSD}_{\mathcal{V}_{TP}}(\mathit{Tr})$ can be conducted analogously. By definition $\mathit{BSI}_{\mathcal{V}_{TP}}(\mathit{Tr})$ holds iff

$$\begin{aligned}&\forall \alpha, \beta \in E^*. \forall c \in \mathit{HI}_{TP}.((\beta.\alpha \in \mathit{Tr} \wedge \alpha|_{\mathit{HI}_{TP}} = \langle\rangle)\\ \Longrightarrow\ &\exists \alpha' \in E^*.(\alpha'|_{L_{TP}} = \alpha|_{L_{TP}} \wedge \alpha'|_{\mathit{HI}_{TP}} = \langle\rangle \wedge \beta.\langle c\rangle.\alpha' \in \mathit{Tr}))\end{aligned}$$

Let C be secure. Given $\alpha, \beta \in E^*$ and $c \in \mathit{HI}_{TP}$ satisfying the conditions above, our aim is to construct an appropriate α' by modifying α.

The proof idea is to construct α' by induction on the length of α, making use of Lemmas 2, 3. The only kind of events classified as HI_{TP} is $\mathit{setvar}(h, \cdot)$-events. Hence, $c = \mathit{setvar}(h, \mathit{val})$ for some val. In the inductive construction, we will prove the following invariant for sequences α and α'. The invariant $\mathit{Inv}(\alpha, \alpha', s, s', i)$ holds iff

$$1.\, \mathit{length}(\alpha) = i \qquad 2.\, \alpha|_{L_{TP}} = \alpha'|_{L_{TP}} \qquad 3.\, \alpha'|_{\mathit{HI}_{TP}} = \langle\rangle \qquad 4.\, s_0 \stackrel{\beta.\alpha}{\Longrightarrow} s$$

$$5.\, s_0 \stackrel{\beta.\langle c\rangle.\alpha'}{\Longrightarrow} s' \text{ (and thereby } \beta.\langle c\rangle.\alpha' \in \mathit{Tr})$$

$$6.\, \forall\, \mathit{tid}.\mathit{thread}_s(\mathit{tid}) = \mathit{thread}_{s'}(\mathit{tid}) = \bot \vee \mathit{thread}_s(\mathit{tid}) = \mathit{thread}_{s'}(\mathit{tid}) = \top$$
$$\vee\, \mathit{thread}_s(\mathit{tid}) = \mathit{thread}_{s'}(\mathit{tid}) = \langle\rangle \vee \mathit{thread}_s(\mathit{tid}) \approxeq_L \mathit{thread}_{s'}(\mathit{tid})$$

$$7.\, \mathit{mem}_s =_L \mathit{mem}_{s'} \qquad 8.\, \mathit{atid}_s = \mathit{atid}_{s'}$$

$$9.\, \mathit{ainfo}_s = \mathit{ainfo}_{s'} \qquad 10.\, \mathit{executed}_s = \mathit{executed}_{s'}$$

Initially, we set $\alpha = \alpha' = \langle\rangle$. Suppose $s_0 \stackrel{\beta.\langle c\rangle}{\Longrightarrow} s'$ for some s'. Clearly, we have $\mathit{Inv}(\alpha, \alpha', s, s', 0)$ since adding $\mathit{setvar}(h, \mathit{val})$ can be done at any time in the computation (the precondition for a $\mathit{setvar}(h, \mathit{val})$-event is *true*). Such an event can only change the value of h in $\mathit{mem}_{s'}$. Lemma 4 guarantees that part 6 of the invariant holds (traversing β preserves the security of the respective commands in the command pool).

In the inductive step, we have to construct α', s' such that $\mathit{Inv}(\alpha, \alpha', s, s', i + 1)$ assuming that $\alpha = \delta.\langle e\rangle$ and, by the induction hypothesis, $\mathit{Inv}(\delta, \delta', t, t', i)$ for some δ', t'. The proof is by considering cases on e such that $s_0 \stackrel{\beta.\delta}{\Longrightarrow} t \stackrel{e}{\longrightarrow} s$ for some s. In all cases we will aim at preserving the invariant. We have $s_0 \stackrel{\beta.\delta}{\Longrightarrow} t$ and $s_0 \stackrel{\beta.\langle c\rangle.\delta'}{\Longrightarrow} t'$. Clearly, e cannot be a $\mathit{setvar}(h, \cdot)$-event by the $\alpha|_{\mathit{HI}_{TP}} = \langle\rangle$ condition on α. The rest of the cases on e are:

setvar$(l, \mathit{\mathbf{val}})$ Set $\alpha' = \delta'.\langle e\rangle$. Then $s_0 \stackrel{\beta.\delta'}{\Longrightarrow} t' \stackrel{e}{\longrightarrow} s'$ for some s'. A $\mathit{setvar}(l, \cdot)$-event is always enabled. The event makes the same update in the low part of the memory in both mem_t and $\mathit{mem}_{t'}$. By induction hypothesis, $\mathit{mem}_t =_L \mathit{mem}_{t'}$ which yields $\mathit{mem}_s =_L \mathit{mem}_{s'}$. The event does not affect any other part of either t or t' (in the transition to s or s' respectively) which gives $\mathit{Inv}(\alpha, \alpha', s, s', i+1)$.

outvar$(l, \mathit{\mathbf{val}})$ Set $\alpha' = \delta'.\langle e\rangle$. The precondition for this event in $t \stackrel{e}{\longrightarrow} s$ is $\mathit{mem}_t = \mathit{val}$. Due to the induction hypothesis $\mathit{mem}_t =_L \mathit{mem}_{t'}$ we have $\mathit{mem}_{t'} = \mathit{val}$ which gives the precondition for $t' \stackrel{e}{\longrightarrow} s'$ for some s'. Thus, $s_0 \stackrel{\beta.\delta'}{\Longrightarrow} t' \stackrel{e}{\longrightarrow} s'$ for some s'. The event does not change any part of either t or t' (in the transition to s or s' respectively) which gives $\mathit{Inv}(\alpha, \alpha', s, s', i + 1)$.

outvar$(h, \mathit{\mathbf{val}})$ Set $\alpha' = \delta'.\langle e'\rangle$ where $e' = \mathit{outvar}(h, \mathit{val}')$ and $\mathit{val}' = \mathit{mem}_{t'}(h)$. Then $s_0 \stackrel{\beta.\delta'}{\Longrightarrow} t' \stackrel{e'}{\longrightarrow} s'$ for some s'. We have $\mathit{Inv}(\alpha, \alpha', s, s', i + 1)$ since the potential difference in val for α' and α does not affect $\alpha'|_{L_{TP}} = \alpha|_{L_{TP}}$ because $\mathit{outvar}(h, \cdot)$ events are classified as $H_{TP} \backslash \mathit{HI}_{TP}$. Note that we could have just as well chosen to set $\alpha' = \delta'$ without affecting the invariant.

schedule$(\mathit{\mathbf{tid}})$ Set $\alpha' = \delta'.\langle e\rangle$. Then $s_0 \stackrel{\beta.\delta'}{\Longrightarrow} t' \stackrel{e}{\longrightarrow} s'$ for some s' since the preconditions for $t' \stackrel{e}{\longrightarrow} s'$ are guaranteed by the induction hypothesis $\mathit{atid}_t = \mathit{atid}_{t'} =$

$\perp$, $live(t', tid) = tt$. The postconditions are updated in the same way, namely, $atid_s = atid_{s'} = tid$ ensuring $Inv(\alpha, \alpha', s, s', i+1)$.

***yield*(*info*)** Set $\alpha' = \delta'.\langle e\rangle$. Yielding *info* in $t \xrightarrow{e} s$ has the precondition $executed_t = tt \wedge ainfo_t = info$. By the induction hypothesis, $executed_t = executed_{t'} = tt \wedge ainfo_t = ainfo_{t'} = info$ which gives $s_0 \overset{\beta.\delta'}{\Longrightarrow} t' \xrightarrow{e} s'$ for some s' The postconditions are updated in the same way: $executed_s = executed_{s'} = ff \wedge ainfo_s = ainfo_{s'} = \perp$ ensuring $Inv(\alpha, \alpha', s, s', i+1)$.

Let us turn to local events. We are able to treat all events at once by appealing to Lemmas 2, 3 and unwinding strong low-bisimulation according to Definition 6. We only have two cases on e:

***first*(*thread*$_t$(*atid*$_t$))** $\neq$ **fork**$(\cdot\cdot)$ We have that $executed_t = ff$, $atid_t \neq \perp$, $live(t, atid_t)$, and $t \xrightarrow{e} s$ hold. By applying the first case of Lemma 2 we receive that $\langle\!| thread_t(atid_t), mem_t |\!\rangle \twoheadrightarrow \langle\!| thread_s(atid_s), mem_s |\!\rangle$. By the induction hypothesis holds $mem_t =_L mem_{t'}$ and $thread_t(atid_t) \approxeq_L thread_{t'}(atid_{t'})$ (note that $atid_t = atid_{t'}$). Unwinding the definition of strong low-bisimulation (Definition 6) yields that there exist C' and mem' such that $\langle\!| thread_{t'}(atid_{t'}), mem_{t'} |\!\rangle \twoheadrightarrow \langle\!| C', mem' |\!\rangle$, $thread_s(atid_s) \approxeq_L C'$, and $mem_s =_L mem'$. According to the induction hypothesis, we have $executed_{t'} = ff$, $atid_{t'} \neq \perp$, and $live(t', atid_{t'})$. Thus, we can apply Lemma 3. By Lemma 3 there exists an event $e' \in E^{\text{MWL}}_{local}$ and an SES-state s' with $t' \xrightarrow{e'} s'$, $atid_{s'} = atid_{t'}$, $thread_{s'}(atid_{s'}) = C'$ and $mem_{s'} = mem'$, $executed_{s'} = tt$. We have $thread_s(atid_s) \approxeq_L thread_{s'}(atid_{s'})$. Moreover, for all tid with $tid \neq atid_{t'}$ holds $thread_{s'}(tid) = thread_{t'}(tid)$. Because $thread_s(atid_s) \approxeq_L thread_{s'}(atid_{s'})$, both $thread_s(atid_s)$ and $thread_{s'}(atid_{s'})$ either make a computation step or terminate. Hence, $terminates(thread_s(atid_s)) = terminates(thread_{s'}(atid_{s'}))$. As a result, *executed* and *ainfo* are updated in the same way for both s and s'. Therefore, setting $\alpha' = \delta'.\langle e'\rangle$ gives $s_0 \overset{\beta.\delta'}{\Longrightarrow} t' \xrightarrow{e'} s'$ such that $Inv(\alpha, \alpha', s, s', i+1)$.

***first*(*thread*$_t$(*atid*$_t$))** $=$ **fork**$(\cdot\cdot)$ We have that $executed_t = ff$, $atid_t \neq \perp$, $live(t, atid_t)$, and $t \xrightarrow{e} s$ hold. By applying Lemma 2 we obtain $\langle\!| thread_t(atid_t), mem_t |\!\rangle \twoheadrightarrow \langle\!| thread_s(atid_t.0) \ldots thread_s(atid_t.n), mem_s |\!\rangle$ where $n \in \mathbb{N}$ is the maximal natural number for which $thread_s(atid_t.n) \neq \perp$ holds. According to the induction hypothesis, we have $mem_t =_L mem_{t'}$ and $thread_t(atid_t) \approxeq_L thread_{t'}(atid_{t'})$ (note that $atid_t = atid_{t'}$). Unwinding the definition of strong low-bisimulation (Definition 6) yields that there exists a command C', a command sequence $\vec{D} = \langle D_1 \ldots D_n\rangle$, and a memory mem' such that $\langle\!| thread_{t'}(atid_{t'}), mem_{t'} |\!\rangle \twoheadrightarrow \langle\!| C'\vec{D}, mem' |\!\rangle$ where $thread_s(atid_t.0) \approxeq_L C'$, $thread_s(atid_t.1) \approxeq_L D_1, \ldots,$ $thread_s(atid_t.n) \approxeq_L D_n$ and $mem_s =_L mem'$. According to the induction hypothesis, we have $executed_{t'} = ff$, $atid_{t'} \neq \perp$, and $live(t', atid_{t'})$. Thus, we can apply Lemma 3. By Lemma 3 there is an event $e' \in E^{\text{MWL}}_{local}$ and an SES-state s' with $t' \xrightarrow{e'} s'$, $atid_{s'} = atid_{t'}.0$, $thread_{s'}(atid_{t'}) = \top$, $mem_{s'} = mem'$, and $executed_{s'} = tt$. Moreover, $thread_{s'}(atid_{t'}.0) = C'$ and $thread_{s'}(atid_{t'}.1) = D_1, \ldots, thread_{s'}(atid_{t'}.n) = D_n$. Further, for all tid with $tid \notin \{atid_{t'}, atid_{t'}.0,$

$\ldots, \mathit{atid}_{t'}.n\}$ holds $\mathit{thread}_{s'}(\mathit{tid}) = \mathit{thread}_{t'}(\mathit{tid})$. Finally, *executed* and *ainfo* are updated in the same way for both s and s' (n threads are spawned in both cases). Therefore, setting $\alpha' = \delta'.\langle e'\rangle$ gives $s_0 \overset{\beta.\delta'}{\Longrightarrow} t' \xrightarrow{e'} s'$ such that $\mathit{Inv}(\alpha, \alpha', s, s', i+1)$. □

Proof. [of Theorem 5] Assuming that $\mathit{MWLPool}(C)$ satisfies $\mathit{BSI}_{\mathcal{V}_{TP}}(\mathit{Tr})$ (where $\mathit{Tr} = \mathit{Tr}_{\mathit{MWLPool}(C)}$) we will show that C is secure, i.e., $C \approxeq_L C$ (by Definition 7). Note that we do not need to assume $\mathit{BSD}_{\mathcal{V}_{TP}}(\mathit{Tr})$ although it is implied by *SecProp*. Let us prove this statement by contraposition. In other words, assuming $C \not\approxeq_L C \wedge \mathit{SecProp}(\mathit{MWLPool}(C))$ we aim to arrive at a contradiction.

By Lemma 6, $C \approxeq_L C \Longleftrightarrow C(\cap_{i<\omega} \approxeq^i_L)C$. Assuming $C \not\approxeq_L C$ implies $\exists i.\, C \not\approxeq^i_L C$. Take $k = \min\{i \mid C \approxeq^i_L C \wedge C \not\approxeq^{i+1}_L C\}$. Note that $k \geq 0$ since, obviously, $C \approxeq^0_L C$. Assume for simplicity that no fork-command occurs in C, i.e., C never spawns new threads. Along the way, we discuss how the proof can be modified to go through without the assumption. We consider two sequences of transitions of the form given in Figure 17. Note that each element of the sequences inherits the command in the configuration from the previous element. Observe that the low parts of the memory progress in both sequences in the same way. The sequences continue as shown in Figure 18. These sequences must exist due to $\forall i \in \{0, \ldots, k\}.\, C \approxeq^i_L C$ and $C \not\approxeq^{k+1}_L C$. Matching the first k steps in both sequences and the low-equivalence of the memories during the first k steps are guaranteed by $\forall i \in \{0, \ldots, k\}.\, C \approxeq^i_L C$. However, at step $k+1$ we have $\forall D_{k+1}, \hat{l}'_{k+1}.\, \langle\!\langle D_k, (h'_k, l_k)\rangle\!\rangle \twoheadrightarrow \langle\!\langle D_{k+1}, (\hat{h}'_{k+1}, \hat{l}'_{k+1})\rangle\!\rangle \Longrightarrow \hat{l}_{k+1} \neq \hat{l}'_{k+1}$. In case C may spawn new threads, the difference is that instead of inheriting the commands from the previous element in the sequences Seq1 and Seq2, the next command is chosen from the command in the previous configuration by selecting the thread that is the counterexample for the low-bisimulation of thread pools obtained at the previous step. Importantly, the sequences of *tid*'s chosen in both Seq1 and Seq2 are then identical. We will use this observation later.

We proceed by constructing two traces of $\mathit{MWLPool}(C)$ that correspond to the two sequences. We will transform one trace into the other using *SecProp* such that the low-equivalences and step matching is preserved. This will take us to a contradiction at step $k+1$. Start off by constructing a trace of $\mathit{MWLPool}(C)$ that corresponds to Seq1. We appeal to Lemma 3 to obtain step-by-step construction of a trace γ of the form given in Figure 19 for some $\mathit{tid}_0, \ldots, \mathit{tid}_k, \mathit{info}_1, \ldots, \mathit{info}_{k+1}$ where each e_i ($i = 1, \ldots, k+1$) is the internal event that corresponds to the $\twoheadrightarrow$-transition in Seq1 according to Lemma 3. In case no threads are spawned $\mathit{tid}_i = 0$ for all $i = 0, \ldots, k$. As we noted, in case C may spawn new threads the sequences of *tid*'s chosen in both Seq1 and Seq2 are identical. By a similar argument the information contained in *info* sequences must also be identical for Seq1 and Seq2 up to info_k. Due to *SecProp* we can insert high events into right tails of γ that do not contain any high events. We get a legitimate trace after the insertion. Let us insert the $\mathit{setvar}(h, \hat{h}_k)$ event between $\mathit{setvar}(h, h_k)$ and e_{k+1} in β. Define $c = \mathit{setvar}(h, \hat{h}_k)$, $\alpha = \langle e_{k+1}.\mathit{yield}(\mathit{info}_{k+1}).\mathit{outvar}(l, \hat{l}_{k+1})\rangle$, and $\beta = \delta.\langle \mathit{setvar}(h, h_k)\rangle$ for some δ such that $\gamma = \beta.\alpha$. We have $\alpha|_{HI_{TP}} = \langle\rangle$. By *SecProp* we have $\exists \alpha'.\, \alpha'|_{L_{TP}} = \alpha|_{L_{TP}} \wedge \alpha'|_{HI_{TP}} = \langle\rangle \wedge \delta.\langle \mathit{setvar}(h, h_k).\mathit{setvar}(h, \hat{h}_k)\rangle.\alpha' \in \mathit{Tr}$. Observe that setting h to $\hat{h}_k$ means restoring the value of h from the result of the

$$\text{Seq1: } \langle\!\langle C, (h_0, l_0) \rangle\!\rangle \rightarrowtriangle \langle\!\langle C_1, (\hat{h}_1, \hat{l}_1) \rangle\!\rangle \quad \langle\!\langle C_1, (h_1, l_1) \rangle\!\rangle \rightarrowtriangle \langle\!\langle C_2, (\hat{h}_2, \hat{l}_2) \rangle\!\rangle \quad \langle\!\langle C_2, (h_2, l_2) \rangle\!\rangle \rightarrowtriangle \ldots$$
$$\text{Seq2: } \langle\!\langle C, (h'_0, l_0) \rangle\!\rangle \rightarrowtriangle \langle\!\langle D_1, (\hat{h}'_1, \hat{l}_1) \rangle\!\rangle \quad \langle\!\langle D_1, (h'_1, l_1) \rangle\!\rangle \rightarrowtriangle \langle\!\langle D_2, (\hat{h}'_2, \hat{l}_2) \rangle\!\rangle \quad \langle\!\langle D_2, (h'_2, l_2) \rangle\!\rangle \rightarrowtriangle \ldots$$

Figure 17: Sequences Seq1 and Seq2

$$\text{Seq1: } \ldots \rightarrowtriangle \langle\!\langle C_k, (\hat{h}_k, \hat{l}_k) \rangle\!\rangle \quad \langle\!\langle C_k, (h_k, l_k) \rangle\!\rangle \rightarrowtriangle \langle\!\langle C_{k+1}, (\hat{h}_{k+1}, \hat{l}_{k+1}) \rangle\!\rangle$$
$$\text{Seq2: } \ldots \rightarrowtriangle \langle\!\langle D_k, (\hat{h}'_k, \hat{l}_k) \rangle\!\rangle \quad \langle\!\langle D_k, (h'_k, l_k) \rangle\!\rangle \not\rightarrowtriangle \langle\!\langle D_{k+1}, (\hat{h}'_{k+1}, \hat{l}_{k+1}) \rangle\!\rangle$$

Figure 18: The continuation of Seq1 and Seq2

$$\begin{aligned}\gamma = \langle & \mathit{schedule}(\mathit{tid}_0).\mathit{setvar}(l, l_0).\mathit{setvar}(h, h_0).e_1.\mathit{yield}(\mathit{info}_1).\mathit{outvar}(l, \hat{l}_1). \\ & \mathit{schedule}(\mathit{tid}_1).\mathit{setvar}(l, l_1).\mathit{setvar}(h, h_1).e_2.\mathit{yield}(\mathit{info}_2).\mathit{outvar}(l, \hat{l}_2).\ldots \\ & \ldots\mathit{schedule}(\mathit{tid}_{k-1}).\mathit{setvar}(l, l_{k-1}).\mathit{setvar}(h, h_{k-1}).e_k.\mathit{yield}(\mathit{info}_k).\mathit{outvar}(l, \hat{l}_k). \\ & \mathit{schedule}(\mathit{tid}_k).\mathit{setvar}(l, l_k).\mathit{setvar}(h, h_k).e_{k+1}.\mathit{yield}(\mathit{info}_{k+1}).\mathit{outvar}(l, \hat{l}_{k+1})\rangle\end{aligned}$$

Figure 19: Sequence γ

$$\begin{aligned}\gamma' = \langle & \mathit{schedule}(\mathit{tid}_0).\mathit{setvar}(l, l_0).e'_1.\mathit{yield}(\mathit{info}_1).\mathit{outvar}(l, \hat{l}_1). \\ & \mathit{schedule}(\mathit{tid}_1).\mathit{setvar}(l, l_1).e'_2.\mathit{yield}(\mathit{info}_2).\mathit{outvar}(l, \hat{l}_2).\ldots \\ & \ldots\mathit{schedule}(\mathit{tid}_{k-1}).\mathit{setvar}(l, l_{k-1}).e'_k.\mathit{yield}(\mathit{info}_k).\mathit{outvar}(l, \hat{l}_k). \\ & \mathit{schedule}(\mathit{tid}_k).\mathit{setvar}(l, l_k).e'_{k+1}.\mathit{yield}(\mathit{info}_{k+1}).\mathit{outvar}(l, \hat{l}_{k+1})\rangle\end{aligned}$$

Figure 20: Sequence γ'

$$\begin{aligned}\gamma'' = \langle & \mathit{schedule}(\mathit{tid}_0).\mathit{setvar}(l, l_0).\mathit{setvar}(h, h'_0).e''_1.\mathit{yield}(\mathit{info}_1).\mathit{outvar}(l, \hat{l}_1). \\ & \mathit{schedule}(\mathit{tid}_1).\mathit{setvar}(l, l_1).\mathit{setvar}(h, h'_1).e''_2.\mathit{yield}(\mathit{info}_2).\mathit{outvar}(l, \hat{l}_2).\ldots \\ & \ldots\mathit{schedule}(\mathit{tid}_{k-1}).\mathit{setvar}(l, l_{k-1}).\mathit{setvar}(h, h'_{k-1}).e''_k.\mathit{yield}(\mathit{info}_k).\mathit{outvar}(l, \hat{l}_k). \\ & \mathit{schedule}(\mathit{tid}_k).\mathit{setvar}(l, l_k).\mathit{setvar}(h, h'_k).e''_{k+1}.\mathit{yield}(\mathit{info}_{k+1}).\mathit{outvar}(l, \hat{l}_{k+1})\rangle\end{aligned}$$

Figure 21: Sequence γ''

$$\text{Seq2: } \langle\!\langle C, (h'_0, l_0) \rangle\!\rangle \rightarrowtriangle \langle\!\langle D_1, (\hat{h}'_1, \hat{l}_1) \rangle\!\rangle \quad \langle\!\langle D_1, (h'_1, l_1) \rangle\!\rangle \rightarrowtriangle \langle\!\langle D_2, (\hat{h}'_2, \hat{l}_2) \rangle\!\rangle \quad \langle\!\langle D_2, (h'_2, l_2) \rangle\!\rangle \rightarrowtriangle \ldots$$
$$\ldots \rightarrowtriangle \langle\!\langle D_k, (\hat{h}'_k, \hat{l}_k) \rangle\!\rangle \quad \langle\!\langle D_k, (h'_k, l_k) \rangle\!\rangle \rightarrowtriangle \langle\!\langle D_{k+1}, (\hat{h}_{k+1}, \hat{l}_{k+1}) \rangle\!\rangle$$

Figure 22: New form of Seq2

previous transition in Seq1. We can just as well omit both updates of h implying $\delta.\alpha' \in \mathit{Tr}$.

Carrying on with the elimination of the rightmost event $\mathit{setvar}(h, h_i)$ for $i = k, \ldots, 0$ we get a trace $\gamma' \in \mathit{Tr}$ that (after removing all occurrences of $\mathit{outvar}(h, \cdot)$-events without loss of generality) has the form depicted in Figure 20. Due to the low events embracing each local event e_i for $i = 1, \ldots, k+1$ it must be the case that there is a one-to-one correspondence between e_i and e'_i for $i = 1, \ldots, k+1$ (although they are not necessarily identical).

The next pass is the insertion of $\mathit{setvar}(h, \cdot)$ following the sequence Seq2. Let us construct new versions of c, α, β in order to apply *SecProp*. Let $c = \mathit{setvar}(h, h'_0)$, $\beta = \langle \mathit{schedule}(\mathit{tid}_0).\mathit{setvar}(l, l_0)\rangle$ and α is such that $\gamma' = \beta.\alpha$. By *SecProp* we have $\exists \alpha'.\, \alpha'|_{L_{TP}} = \alpha|_{L_{TP}} \wedge \alpha'|_{HI_{TP}} = \langle\rangle \wedge \beta.\langle \mathit{setvar}(h, h'_0)\rangle.\alpha' \in \mathit{Tr}$. Continuing rightmost $\mathit{setvar}(h, h'_i)$ $(i = 0, \ldots, k+1)$ insertion we get a trace $\gamma'' \in \mathit{Tr}$ that (again, after removing all occurrences of $\mathit{outvar}(h, \cdot)$-events without loss of generality) has the form depicted in Figure 21. Due to the low *schedule*- and *yield*-events embracing each local event e'_i $(i = 1, \ldots, k+1)$ it must be the case that there is a one-to-one correspondence between e'_i and e''_i for $i = 1, \ldots, k+1$ (although they are not necessarily identical).

According to Lemma 2 we can now convert the trace γ'' into a sequence of $\twoheadrightarrow$-transitions. The crucial property is that these transitions are deterministic, i.e., if $\langle\!\langle E, \mathit{mem}\rangle\!\rangle \twoheadrightarrow \langle\!\langle E', \mathit{mem}'\rangle\!\rangle$ then $\forall E'', \mathit{mem}''.\, \langle\!\langle E, \mathit{mem}\rangle\!\rangle \twoheadrightarrow \langle\!\langle E'', \mathit{mem}''\rangle\!\rangle \implies E' = E'' \wedge \mathit{mem}' = \mathit{mem}''$. In case C may spawn threads, we also need the observation we made about the same sequences of *tid*'s and *info*'s that are used in the construction of Seq1 and Seq2. This is important to restore the branching behavior of traces as it was in Seq1 and Seq2. The fact that only programs with the same branching structure can be low-bisimilar is reflected in the traces, because the branching behavior is recorded in the low *schedule*-events. Thus, by induction, we can restore the sequence Seq2, as depicted in Figure 22 for some command D_{k+1}, which contradicts our original assumption about Seq2. □

Proof. [Sketch of Lemma 7] The proof can be carried out along the same lines as the proof of Lemma 1. Choose $\gamma \in E^*$ with $s_0 \stackrel{\gamma}{\Longrightarrow} s$. The proof is by induction on γ. The step case ($\gamma = \delta.\langle e\rangle$), is trivial if e is a *setvar*-, *outvar*, input *trans*-event or $e \notin E^{\mathit{pid}}$. For other events, a case distinction is made according to the induction hypothesis where $e \in E^{\text{DMWL}}_{\mathit{local}}$, $e = \mathit{trans}(\mathit{cid}, \mathit{val})$ for some $\mathit{cid} \in \mathit{CID}$ and $\mathit{val} \in \mathit{VAL}$ with $\mathit{sender}(\mathit{cid}) = \mathit{pid}$, $e = \mathit{yield}(\mathit{info})$, $e = \mathit{schedule}(\mathit{tid})$, respectively, hold in the four cases. Each of these cases follows from Figures 2, 7, 15, and 16. □

Proof. [Sketch of Lemma 8] The proof can be carried out along the same lines as for Lemma 2. A case distinction on e is made. The cases where $e \in E^{\text{MWL}}_{\mathit{local}}$ can be handled as in the proof of Lemma 2 because $\mathit{channel}(s') = \mathit{channel}(s)$. If e is a *send*-, *receive*-, $\mathit{ite\text{-}rcv}^{\mathit{tt}}$-, or $\mathit{ite\text{-}rcv}^{\mathit{ff}}$-event then the proposition follows from Figures 9 and 16. □

Proof. [of Theorem 6] First, we establish a few restrictions on γ. Observe that the following properties are satisfied for all $\mathit{cid}, \mathit{cid}' \in \mathit{CID}$, all $\mathit{val}, \mathit{val}' \in \mathit{VAL}$, all $\mathit{pid}_r, \mathit{pid}_i \in \mathit{Pid}$ with $\mathit{pid}_r \neq \mathit{pid}_i$, and all $\mathit{schedule}^{\mathit{pid}_r}(\ldots)$, $\mathit{ite\text{-}rcv}^{\mathit{ff}^{\mathit{pid}_r}}(\mathit{cid}, \ldots)$, $\mathit{send}^{\mathit{pid}_i}(\mathit{cid}, \mathit{val}')$-events:

- If $\gamma = \gamma_1.\langle \mathit{schedule}^{pid_r}(\dots)\rangle.\gamma_2.\langle \mathit{ite\text{-}rcv}^{\mathit{ff}\,pid_r}(cid,\dots)\rangle.\gamma_3$
and $\gamma_2|_{E^{pid_r}} \in \{\mathit{trans}(cid', val') \mid \mathit{receiver}(cid') = pid_r, \mathit{sender}(cid') \neq pid_r\}^*$
then $s \overset{\gamma'}{\Longrightarrow} s'$ where $\gamma' = \gamma_1.\langle \mathit{schedule}^{pid_r}(\dots).\mathit{ite\text{-}rcv}^{\mathit{ff}\,pid_r}(cid,\dots)\rangle.\gamma_2.\gamma_3$.

- If $\gamma = \gamma_1.\langle \mathit{send}^{pid_i}(cid, val)\rangle.\gamma_2.\langle \mathit{schedule}^{pid_r}(\dots).\mathit{ite\text{-}rcv}^{\mathit{ff}\,pid_r}(cid,\dots)\rangle.\gamma_3$
and $\gamma_2|_{E^{pid_i}} \in \{\mathit{trans}(cid', val') \mid cid' \neq cid\}^*$
then $s \overset{\gamma'}{\Longrightarrow} s'$ where
$\gamma' = \gamma_1.\gamma_2.\langle \mathit{schedule}^{pid_r}(\dots).\mathit{ite\text{-}rcv}^{\mathit{ff}\,pid_r}(cid,\dots).\mathit{send}^{pid_i}(cid, val)\rangle.\gamma_3$.

Both of these properties can be proved by a simple induction over γ_2. According to the above properties we may make the following assumptions about γ. These assumptions will be helpful for applying Lemma 8.

1. If $\gamma = \gamma_1.\langle \mathit{schedule}^{pid_r}(\dots)\rangle.\gamma_2.\langle \mathit{ite\text{-}rcv}^{\mathit{ff}\,pid_r}(cid,\dots)\rangle.\gamma_3$
and $\gamma_2|_{E^{pid_r}} \in \{\mathit{trans}(cid', val') \mid \mathit{receiver}(cid') = pid_r \wedge \mathit{sender}(cid') \neq pid_r\}^*$
then $\gamma_2 = \langle\rangle$.

2. If $\gamma = \gamma_1.\langle \mathit{send}^{pid_i}(cid, val)\rangle.\gamma_2.\langle \mathit{schedule}^{pid_r}(\dots).\mathit{ite\text{-}rcv}^{\mathit{ff}\,pid_r}(cid,\dots)\rangle.\gamma_3$
then an event $\mathit{trans}(cid, val)$ occurs in γ_2. Together with the precondition of $\mathit{ite\text{-}rcv}^{\mathit{ff}}$, this implies that there must be a *receive*- or $\mathit{ite\text{-}rcv}^{\mathit{tt}}$-event in γ_2 that consumes the message *val* on *cid*.

Moreover, we assume that γ contains no *setvar*- or *outvar*-events. That γ contains no *setvar*-events is an assumption of the theorem. *outvar*-events have no effect on the state and, hence, they can safely be removed.

The proof of the theorem proceeds by induction on the length of γ.

Base case ($\gamma = \langle\rangle$): The proposition holds because $s' = s$ and $\twoheadrightarrow^*$ is reflexive.

Step case ($\gamma = \langle e\rangle.\delta$): There exists $s_i \in S$ with $s \xrightarrow{e} s_i$ and $s_i \overset{\delta}{\Longrightarrow} s'$. We make a case distinction depending on if e is associated with one or more DMWL thread pools (at most two).

Firstly, assume that there are $pid_i, pid_r \in \mathit{Pid}$ with $pid_i \neq pid_r$ and $e \in E^{pid_i} \cap E^{pid_r}$. $e = \mathit{trans}(cid, val)$ holds. Without loss of generality, assume $\mathit{sender}(cid) = pid_i$ and $\mathit{receiver}(cid) = pid_r$. We have $\mathsf{channel}(s') = \mathsf{channel}(s)$. The occurrence of e affects only *inbuf* and *outbuf* but no other state variables. The proposition follows from the reflexivity of $\twoheadrightarrow^*$.

Secondly, assume that there is exactly one $pid \in \mathit{Pid}$ with $e \in E^{pid}$. Let $s|_{pid} = (\mathit{mem}_s, \mathit{thread}_s, \mathit{atid}_s, \dots, \mathit{executed}_s, \dots, \mathit{outbuf}_s)$. We make a case distinction according to Lemma 1.

- Assume $\mathit{executed}_s = \mathit{ff} \wedge \mathit{atid}_s \neq \bot \wedge \mathit{thread}_s(\mathit{atid}_s) \notin \{\bot, \top, \langle\rangle\} \wedge \mathit{outbuf}_s = \langle\rangle$. The proposition follows from $e \in E^{\text{DMWL}}_{\mathit{local}}$, Lemma 8, the frame-axioms for events in $E^{\text{DMWL}}_{\mathit{local}}$, Definition 20, rule Step, rule Pick, and the induction hypothesis. Note that if $e = \mathit{ite\text{-}rcv}^{\mathit{ff}}(cid', \dots)$ then $\mathsf{channel}(s)(cid') = \langle\rangle$ holds because of our initial assumptions (1,2) on γ. Hence, the requirements of Lemma 8 are, indeed, satisfied.

- The case $executed_s = tt \wedge atid_s \neq \bot \wedge outbuf_s \neq \langle\rangle$ cannot happen. Since $e = trans(cid, val)$, $sender(cid), receiver(cid) \in Pid$, and a process cannot send to itself, this case contradicts our assumption that there is exactly one $pid \in Pid$ with $e \in E^{pid}$.
- Assume $executed_s = tt \wedge atid_s \neq \bot \wedge outbuf_s = \langle\rangle$. $e = yield(ainfo_s)$ holds. $config(s', pid) = (cseq(thread_s), mem_s)$ and $channel(s') = channel(s)$ are implied by the postcondition of *yield*. Proposition follows from the induction hypothesis.
- Assume $executed_s = ff \wedge atid_s = \bot \wedge outbuf_s = \langle\rangle$. $e = schedule(tid)$ holds. $config(s', pid) = (cseq(thread_s), mem_s)$ and $channel(s') = channel(s)$ are implied by the postcondition of *schedule*. Proposition follows from the induction hypothesis. ☐

Proof. [Sketch of Lemma 9] The proof can be carried out along the same lines as for Lemma 3. Let $\mathcal{D}$ be a derivation of $\langle\!|thread_s(atid_s), mem_s, channel(s)|\!\rangle \twoheadrightarrow \langle\!|C', mem', \sigma'|\!\rangle$ (or $\langle\!|thread_s(atid_s), mem_s, channel(s)|\!\rangle \twoheadrightarrow \langle\!|C' D_1 \ldots D_n, mem', \sigma'|\!\rangle$). There must be exactly one application of one of the rules Skip, Assign, If_{tt}, If_{ff}, While_{tt}, While_{ff}, Fork, Send, Receive, IfRcv_{tt}, or IfRcv_{ff} in $\mathcal{D}$. All cases except for Send, Receive, IfRcv_{tt}, or IfRcv_{ff} can be handled as in the proof of Lemma 3 because $\sigma' = channel(s)$. The remaining cases follow from Figures 9 and 16. ☐

Proof. [Sketch of Theorem 7] The proof can be carried out along the same lines as the proof of Theorem 3. We only point out the differences.

The induction needs to be done over the application of rule Step rather than rule Pick. In the step case, Lemma 9 is used rather than Lemma 3. For the case where the command is a send-command, choose $\gamma = \langle e_1.e_2.e_3.e_4\rangle$ where $e_1 = schedule(\ldots)$, $e_2 \in E^{\text{DMWL}}_{local}$, $e_3 = trans(\ldots)$, $e_4 = yield(\ldots)$. For the case where the command is no send-command, choose $\gamma = \langle e_1.e_2.e_4\rangle$. If e_2 is a *receive*- or a $ite\text{-}rcv^{tt}$-event, the requirement $channel(s)(cid') = \langle\rangle$ of Lemma 9 is, indeed, satisfied if $pending_s(cid') = \langle\rangle$. The argument is that *inbuf* of the chosen thread pool is empty after the *schedule*-event and *outbuf* is empty for all thread pools (follows from reachability of s, $atid_s = \bot$, $executed_s = ff$, and Lemma 7). ☐

Proof. [of Lemma 10] The proof is conducted similarly to the proof of Lemma 4. We proceed by induction on the length of β, using Lemma 8 to match the transition in the semantics of DMWL, and unwinding Definition 10 of strong low-bisimulation. In the inductive step, we have $s_0 \overset{\delta}{\Longrightarrow} t \overset{e}{\longrightarrow} s$ such that $\forall tid.\, live(t, tid) \Longrightarrow thread_t(tid) \approxeq_L thread_t(tid)$. Suppose either $first(thread_t(atid_t)) = \mathsf{fork}(\cdots)$ or $first(thread_t(atid_t)) \neq \mathsf{fork}(\cdots)$ such that if $e = ite\text{-}rcv^{ff}(cid, var, val, C_1, C_2)$ then $channel(t)(cid) = \langle\rangle$. Then, Lemma 8 is directly applicable and the proof follows the proof of Lemma 4.

In case $e = ite\text{-}rcv^{ff}(cid, var, val, C_1, C_2)$ and $channel(t)(cid) \neq \langle\rangle$, we observe that $pending_t(cid) = \langle\rangle$ by the precondition for e. Because $receiver(cid) = pid$, we have $channel(t)(cid) = inbuf_t(cid).pending_t(cid)$ (cf. Section 7.6). Thus, $inbuf_t(cid) \neq \langle\rangle$. The precondition for e requires that $first(thread_t(atid_t)) = \mathsf{if\text{-}receive}(cid, var, C_1, C_2)$. Moreover, we have the DMWL transition $\langle\!|thread_t(atid_t), mem_t, channel(t)[cid \mapsto \langle\rangle]|\!\rangle \twoheadrightarrow \langle\!|thread_s(atid_s), mem_s, channel(s)[cid \mapsto \langle\rangle]|\!\rangle$ such that $first(thread_s(atid_s)) = C_2$,

$mem_t = mem_s$, and $\mathit{channel}(t) = \mathit{channel}(s)$. Now, $thread_t(atid_t)$ is secure by the induction hypothesis. By Definition 10 of strong low-bisimulation, there exist C', mem', and σ' such that $\langle\!| thread_t(atid_t), mem_t, \mathit{channel}(t)[cid \mapsto \langle\rangle] |\!\rangle \twoheadrightarrow \langle\!| C', mem', \sigma' |\!\rangle$ where $mem_s =_L mem'$, $thread_s(atid_s) \cong_L C'$, and $\mathit{channel}(s)[cid \mapsto \langle\rangle] =_L \sigma'$. Since $\twoheadrightarrow$-transitions are deterministic, it must be the case that $C' = thread_s(atid_s)$, $mem' = mem_s$ and $\sigma' = \mathit{channel}(s)[cid \mapsto \langle\rangle]$. Thus, $thread_s(atid_s) \cong_L thread_s(atid_s)$. Because no other *tid*'s are affected, the proof for this case is completed. $\square$

Proof. [Sketch of Theorem 8] Assume that C is secure. We need to show that the DMWL process $DMWLProcess(pid, C)$ satisfies *SecProp*. According to Section 3.2, we need to prove $BSI_{\mathcal{V}_{TP}}(Tr)$ and $BSD_{\mathcal{V}_{TP}}(Tr)$ where $Tr = Tr_{MWLPool(C)}$. We prove $BSI_{\mathcal{V}_{TP}}(Tr)$ ($BSD_{\mathcal{V}_{TP}}(Tr)$ is proved similarly) using the technique of Theorem 4. Given $\alpha, \beta \in E^*$ and $c \in HI_{TP}$ where c is to be inserted, our aim is to inductively construct an appropriate α' by modifying α. In the inductive construction for sequences α,α', we will prove the invariant $Inv(\alpha, \alpha', s, s', i)$ from the proof of Theorem 4, extended with the following requirements:

$$11.\, blocked\text{-}set(s) = blocked\text{-}set(s')$$
$$12.\, inbuf_s =_L inbuf_{s'} \wedge pending_s =_L pending_{s'}$$
$$13.\, \forall cid \in CID.\ dom_{ch}(cid) = low \Longrightarrow$$
$$(outbuf_s = (cid, val) \Longleftrightarrow outbuf_{s'} = (cid, val))$$

Lemma 10 guarantees that part 6 of the invariant holds for the base case. In the inductive step, we have to construct α', s' such that $Inv(\alpha, \alpha', s, s', i+1)$ assuming that $\alpha = \delta.\langle e\rangle$ and, by the induction hypothesis, $Inv(\delta, \delta', t, t', i)$ for some δ', t'. The proof is by considering cases on e such that $s_0 \overset{\beta.\delta}{\Longrightarrow} t \xrightarrow{e} s$ for some s. In all cases we will aim at preserving the invariant. We have $s_0 \overset{\beta.\delta}{\Longrightarrow} t$ and $s_0 \overset{\beta.\langle c\rangle.\delta'}{\Longrightarrow} t'$. Clearly, e cannot be a $setvar(h, \cdot)$ or $trans(cid, \cdot)$ (where $dom_{ch}(cid) = high$ and $receiver(cid) = pid$) by the $\alpha|_{HI_{TP}} = \langle\rangle$ condition on α. The cases $e = setvar(l, val)$, $e = outvar(l, val)$, and $e = outvar(h, val)$ are handled identically to the cases in the proof of Theorem 4. In case $e = schedule(tid)$, part 11 of the invariant ensures that we can schedule the same thread in t'. Part 12 is updated in both s and s' in the same way. The case $e = yield(info, blocked\text{-}info)$ is treated similarly. The cases when e is a new *trans*-event are:

trans*(*cid*, *val*), *receiver*(*cid*) = *pid It must be the case that $dom_{ch}(cid) = low$ because $dom_{ch}(cid) = high$ would imply $trans(cid, val) \in HI_{TP}$ and e cannot be such an event. The case is analogous to the one for $setvar(l, val)$-events (the precondition is always *true*).

trans*(*cid*, *val*), *sender*(*cid*) = *pid The precondition for this event is $outbuf(t, cid) = (cid, val)$. In case $dom_{ch}(cid) = low$, due to part 13 of the invariant, we have $outbuf(t', cid) = (cid, val)$, which enables e in state t'. Set $\alpha' = \delta'.\langle e\rangle$; and we have $outbuf(s, cid) = outbuf(s', cid) = \langle\rangle$ which preserves the invariant.

In case $dom_{ch}(cid) = high$, there are two different cases: If $outbuf(t', cid) = (cid, val')$, then $\alpha' = \delta'.\langle e'\rangle$ where $e' = trans(cid, val')$ which preserves the invariant. If *outbuf*$(t', cid) = \langle\rangle$, then take simply $\alpha' = \delta'$ which also preserves the invariant.

The cases on local events are proved along the lines of the proof of Theorem 4. The cases when Lemmas 8, 9 are not applicable are resolved separately in the same manner as in the proof of Lemma 10. Otherwise, we apply Lemmas 8, 9 and unwind strong low-bisimulation according to Definition 10. Part 11 of the invariant is preserved as a corollary of the following general statement. For a channel *cid* and commands D and D' such that $D \approxeq_L D'$, if $\mathit{first}(D) = \mathsf{receive}(\mathit{cid}, \cdot)$ then $\mathit{first}(D') = \mathsf{receive}(\mathit{cid}, \cdot)$ and, furhermore, $\mathit{dom}_{ch}(\mathit{cid}) = \mathit{low}$. Due to this statement and parts 6 and 12 of the invariant, part 11 is preserved through the case when e is a local event. $\square$

Proof. [Sketch of Theorem 9] The proof is conducted by contraposition, as for Theorem 5. We will use a version of Lemma 6 for the new strong low-bisimulation to facilitate the proof. Assuming $C \not\approxeq_L C \wedge \mathit{SecProp}(\mathit{DMWLProcess}(\mathit{pid}, C))$ we arrive at a contradiction to $\mathit{BSI}_{\mathcal{V}_{TP}}(\mathit{Tr}) \wedge \mathit{BSD}_{\mathcal{V}_{TP}}(\mathit{Tr})$ (where $\mathit{Tr} = \mathit{Tr}_{\mathit{MWLPool}(C)}$).

In contrast to the proof of Theorem 5, we do rely on $\mathit{BSD}_{\mathcal{V}_{TP}}(\mathit{Tr})$ in the proof. The idea is to use Lemmas 8 and 9 in building the sequences similar to the ones in the proof of Theorem 5. Assume (without loss of generality) we only have two high channels cid_h ($\mathit{receiver}(\mathit{cid}_h) = \mathit{pid}$) and cid_h^o ($\mathit{sender}(\mathit{cid}_h^o) = \mathit{pid}$) and two low channels cid_l ($\mathit{receiver}(\mathit{cid}_l) = \mathit{pid}$) and cid_l^o ($\mathit{sender}(\mathit{cid}_l^o) = \mathit{pid}$). We treat $\mathit{trans}(\mathit{cid}_h, \cdot)$-events similarly to $\mathit{setvar}(h, \cdot)$-events and $\mathit{trans}(\mathit{cid}_h^o, \cdot)$-events similarly to local events. Both $\mathit{trans}(\mathit{cid}_l, \cdot)$- and $\mathit{trans}(\mathit{cid}_l^o, \cdot)$-events are similar to other low events.

The sequences Seq1 and Seq2 consist now of configurations extended with channel status functions that only differ in the high part for each respective configuration. When building a trace γ of $\mathit{DMWLProcess}(\mathit{pid}, C)$ that corresponds to Seq1 we use Lemma 9. We use the same idea as in the proof of Theorem 5 adding now *trans*-events that populate the channels prior to *schedule*-events and also *trans*-events before *yield*-events that clear *outbuf* if a local *send*-event has occurred. In the example below, let us represent the channel status function as a quadruple of sequences where the first and second sequences correspond to the high and low input channels, respectively; and the third and fourth sequences correspond to the high and low output channels, respectively. The first elements of the sequences are:

$$\text{Seq1:} \langle\!|C, (h_0, l_0), (\vec{v}_0, \vec{w}_0, \vec{x}_0, \vec{y}_0)|\!\rangle \twoheadrightarrow \langle\!|C_1, (\hat{h}_1, \hat{l}_1), (\vec{\hat{v}}_1, \vec{\hat{w}}_1, \vec{\hat{x}}_1, \vec{\hat{y}}_1)|\!\rangle$$
$$\text{Seq2:} \langle\!|C, (h'_0, l_0), (\vec{v'}_0, \vec{w}_0, \vec{x'}_0, \vec{y}_0)|\!\rangle \twoheadrightarrow \langle\!|D_1, (\hat{h}'_1, \hat{l}_1), (\vec{\hat{v}}'_1, \vec{\hat{w}}_1, \vec{\hat{x}}'_1, \vec{\hat{y}}_1)|\!\rangle$$

First, we transform each transition of the sequences discarding redundant outputs and all inputs except those consumed during the transition. Let us illustrate the transformation by an example of two first steps of the following two sequences. The first transitions of these sequences are:

$$\text{Seq1:} \langle\!|C, (h_0, l_0), (v_0^1 v_0^2, w_0^1, x_0^1, y_0^1 y_0^2)|\!\rangle \twoheadrightarrow \langle\!|C_1, (\hat{h}_1, \hat{l}_1), (v_0^1 v_0^2, w_0^1, x_0^1 x_0^2, y_0^1 y_0^2)|\!\rangle$$
$$\text{Seq2:} \langle\!|C, (h'_0, l_0), ({v'}_0^1 {v'}_0^2, w_0^1, \langle\rangle, y_0^1 y_0^2)|\!\rangle \twoheadrightarrow \langle\!|D_1, (\hat{h}'_1, \hat{l}_1), ({v'}_0^1, w_0^1, \langle\rangle, y_0^1 y_0^2)|\!\rangle$$

Note that the low-level components are matched (including the low-input and low-output sequences) in the respective configurations. At the first step, no inputs are consumed in Seq1, while only a high input is consumed in Seq2. We can safely remove

w_0^1 from the low input of both sequences and $v_0^1 v_0^2$ and ${v'}_0^1$ from the high input of Seq1 and Seq2, respectively. No output is produced by Seq2, while only a high output is produced by Seq1. Hence, we can safely discard $y_0^1 y_0^2$ from the low output of both sequences and x_0^1 from the high output of Seq1. The first transitions of the resulting sequences Seq1′ and Seq2′ are:

$$\text{Seq1}'\text{: } \langle\!| C, (h_0, l_0), (\langle\rangle, \langle\rangle, \langle\rangle, \langle\rangle) |\!\rangle \rightharpoondown \langle\!| C_1, (\hat{h}_1, \hat{l}_1), (\langle\rangle, \langle\rangle, x_0^2, \langle\rangle) |\!\rangle$$
$$\text{Seq2}'\text{: } \langle\!| C, (h'_0, l_0), ({v'}_0^2, \langle\rangle, \langle\rangle, \langle\rangle) |\!\rangle \rightharpoondown \langle\!| D_1, (\hat{h}'_1, \hat{l}_1), (\langle\rangle, \langle\rangle, \langle\rangle, \langle\rangle) |\!\rangle$$

Suppose the second transitions of the sequences Seq1 and Seq2 are:

$$\text{Seq1: } \langle\!| C_1, (h_1, l_1), (v_1^1, w_1^1 w_1^2, \langle\rangle, \langle\rangle) |\!\rangle \rightharpoondown \langle\!| C_2, (\hat{h}_2, \hat{l}_2), (v_1^1, w_1^1, \langle\rangle, \langle\rangle) |\!\rangle$$
$$\text{Seq2: } \langle\!| D_1, (h'_1, l_1), ({v'}_1^1 {v'}_1^2, w_1^1 w_1^2, \langle\rangle, \langle\rangle) |\!\rangle \rightharpoondown \langle\!| D_2, (\hat{h}'_2, \hat{l}_2), ({v'}_1^1 {v'}_1^2, w_1^1, \langle\rangle, \langle\rangle) |\!\rangle$$

At the second step, only low inputs are consumed in both Seq1 and Seq2. Thus, we can safely remove v_1^1 and ${v'}_1^1 {v'}_1^2$ from the high inputs of Seq1 and Seq2, respectively. Also, we can remove w_1^1 from the low inputs of both Seq1 and Seq2. On the other hand, we can safely insert x_0^2 from the first step of Seq1′ in the high output sequence for Seq1′ for the second step. The second transitions of the resulting sequences Seq1′ and Seq2′ are:

$$\text{Seq1}'\text{: } \langle\!| C_1, (h_1, l_1), (\langle\rangle, w_1^2, x_0^2, \langle\rangle) |\!\rangle \rightharpoondown \langle\!| C_2, (\hat{h}_2, \hat{l}_2), (\langle\rangle, \langle\rangle, x_0^2, \langle\rangle) |\!\rangle$$
$$\text{Seq2}'\text{: } \langle\!| D_1, (h'_1, l_1), (\langle\rangle, w_1^2, \langle\rangle, \langle\rangle) |\!\rangle \rightharpoondown \langle\!| D_2, (\hat{h}'_2, \hat{l}_2), (\langle\rangle, \langle\rangle, \langle\rangle, \langle\rangle) |\!\rangle$$

The beginning of γ that corresponds to the first two steps of Seq1′ is then:

$$\begin{aligned}\gamma = \langle & \mathit{schedule}(\mathit{tid}_0).\mathit{setvar}(l, l_0).\mathit{setvar}(h, h_0). \\ & e_1.\mathit{trans}(\mathit{cid}_h^o, x_0^2).\mathit{yield}(\mathit{info}_1).\mathit{outvar}(l, \hat{l}_1). \\ & \mathit{trans}(\mathit{cid}_l, w_1^2).\mathit{schedule}(\mathit{tid}_0).\mathit{setvar}(l, l_1).\mathit{setvar}(h, h_1). \\ & e_2.\mathit{yield}(\mathit{info}_2).\mathit{outvar}(l, \hat{l}_2) \ldots \rangle\end{aligned}$$

In the example above, e_1 is a $\mathit{send}(\mathit{cid}_h^o, x_0^2)$-event, and, therefore, it is followed by a $\mathit{trans}(\mathit{cid}_h^o, x_0^2)$-event. The rest of the proof uses Lemma 8 and continues along the lines of the proof Theorem 5. Note that Lemmas 8 and 9 are applicable to the traces we create, because input *trans*-events appear before *schedule*-events that ensure that all values are forwarded from *inbuf* to *pending* of the current state. Note that $\mathit{BSD}_{\mathcal{V}_{TP}}(\mathit{Tr})$ is used during the first pass (cf. sequence γ' in the proof of Theorem 5) in order to delete the $\mathit{trans}(\mathit{cid}_h, v_i^j)$-events, whereas $\mathit{BSI}_{\mathcal{V}_{TP}}(\mathit{Tr})$ is used during the second pass (cf. sequence γ'' in the proof of Theorem 5) in order to insert the $\mathit{trans}(\mathit{cid}_h, {v'}_i^j)$-events. Thus, we make use of both basic security predicates in this proof. □

Proof. [of Lemma 11] The proof is carried out along the same lines as the proof of the Zipping Lemma for forward correctability in [20]. Because of $\mathit{BSD}_{\mathcal{V}^{pid}}(\mathit{Tr}^{pid})$, it suffices to prove the lemma under the assumption $t^{pid}|_{C^{pid}} = \langle\rangle$ for all $pid \in \mathit{Pid}$.

The proof proceeds by induction on λ. In the base case, i.e., for $\lambda = \langle\rangle$, the proposition holds with the choice $t = \langle\rangle$. In the step case, i.e., for $\lambda = \langle v \rangle.\lambda'$, we make a case distinction. Firstly, assume that there are $pid_i, pid_r \in Pid$ (with $pid_i \neq pid_r$) such that $v \in V^{pid_i} \cap V^{pid_r}$. Secondly, assume that there is exactly one $pid' \in Pid$ for which $V^{pid'}$ holds. We only prove the first case, since this is the more difficult case.

Since DMWL processes may only interact via *trans*-events, $v = trans(cid, val)$ must hold. Without loss of generality, $sender(cid) = pid_i$ and $receiver(cid) = pid_r$. Choose $r_1^{pid_i}, s_1^{pid_i} \in E^{pid_i *}$ and $r_1^{pid_r}, s_1^{pid_r} \in E^{pid_r *}$ such that $t^{pid_i} = r_1^{pid_i}.\langle v\rangle.s_1^{pid_i}$, $r_1^{pid_i}|_V = \langle\rangle$, $t^{pid_r} = r_1^{pid_r}.\langle v\rangle.s_1^{pid_r}$, and $r_1^{pid_r}|_V = \langle\rangle$ hold. Because $(r_1^{pid_r}|_{E^{pid_i}}) \in C^{pid_i *}$ and $BSI_{\mathcal{V}^{pid_i}}(Tr^{pid_i})$, there are $r_2^{pid_i}, s_2^{pid_i} \in E^{pid_i *}$ with $\tau|_{E^{pid_i}}.(r_1^{pid_r}|_{E^{pid_i}}).r_2^{pid_i}.\langle v\rangle.s_2^{pid_i} \in Tr^{pid_i}$, $r_2^{pid_i}|_V = \langle\rangle$, $s_2^{pid_i}|_V = s_1^{pid_i}|_V$, and $(r_2^{pid_i}.\langle v\rangle.s_2^{pid_i})|_{C^{pid_i}} = \langle\rangle$. Since $v \in V^{pid_r} \cap \nabla^{pid_r}$, $(r_2^{pid_i}|_{E^{pid_r}}) \in (C^{pid_r} \cap \Upsilon^{pid_r})^*$, and $FCI_{\mathcal{V}^{pid_r}}^{\nabla^{pid_r}, \Upsilon^{pid_r}}(Tr^{pid_r})$, there exists a sequence $s_2^{pid_r} \in E^{pid_r *}$ with $\tau|_{E^{pid_r}}.r_1^{pid_r}.(r_2^{pid_i}|_{E^{pid_r}}).\langle v\rangle.s_2^{pid_r} \in Tr^{pid_r}$, $s_2^{pid_r}|_V = s_1^{pid_r}|_V$, and $s_2^{pid_r}|_{C^{pid_r}} = \langle\rangle$. For each $pid \in Pid \setminus \{pid_i, pid_r\}$, there is a $s_2^{pid} \in E^{pid*}$ with $(r_1^{pid_r}.r_2^{pid_i})|_{E^{pid}}.s_2^{pid} \in Tr^{pid}$, $s_2^{pid}|_V = t^{pid}|_V$, and $s_2^{pid}|_{C^{pid}} = \langle\rangle$ (because $(r_1^{pid_r}.r_2^{pid_i})|_{E^{pid}} \in C^{pid*}$ and $BSI_{\mathcal{V}^{pid}}(Tr^{pid})$). The proposition follows after an application of the induction hypothesis for $\tau.r_1^{pid_r}.r_2^{pid_i}.\langle v\rangle$, λ', and s_2^{pid} (for $pid \in Pid$). □

Proof. [of Theorem 10] We abbreviate by $\nabla = \nabla^{Pid}$ and $\Upsilon = \Upsilon^{Pid}$. We have to show that $BSD_{\mathcal{V}}(Tr)$, $BSI_{\mathcal{V}}(Tr)$, and $FCI_{\mathcal{V}}^{\nabla,\Upsilon}(Tr)$ hold. We only prove $FCI_{\mathcal{V}}^{\nabla,\Upsilon}(Tr)$ explicitly. The other statements can be proved along the same lines.

Assume $\alpha, \beta \in E^*$ and $c, v \in E$ such that $\beta.\langle v\rangle.\alpha \in Tr$, $c \in C \cap \Upsilon$, $v \in V \cap \nabla$, and $\alpha|_C = \langle\rangle$. According to the definition of composition, we have $(\beta.\langle v\rangle.\alpha)|_{E^{pid}} \in Tr^{pid}$. Our restrictions on the composition on DMWL processes together with $c \in C \cap \Upsilon$ imply that $c \in E^{pid_c}$ holds for exactly one $pid_c \in Pid$. Our restrictions on the composition on DMWL processes together with $v \in V \cap \nabla$ imply that $v \in E^{pid_v}$ holds for exactly one $pid_v \in Pid$. We make a case distinction on $pid_c = pid_v \vee pid_c \neq pid_v$.

Firstly, assume $pid_c = pid_v$. A trace $\alpha^{pid_c} \in E^{pid_c *}$ exists with $\beta|_{E^{pid_c}}.\langle c.v\rangle.\alpha^{pid_c} \in Tr^{pid_c}$, $\alpha^{pid_c}|_V = \alpha|_{V^{pid_c}}$, and $\alpha^{pid_c}|_{C^{pid_c}} = \langle\rangle$ because $BSD_{\mathcal{V}^{pid_c}}(Tr^{pid_c})$ (delete $\alpha|_{C^{pid_c}}$ in $(\beta.\langle v\rangle.\alpha)|_{E^{pid_c}}$) and $FCI_{\mathcal{V}^{pid_c}}^{\nabla,\Upsilon}(Tr^{pid_c})$ (insert c before v). The assumptions of the Zipping Lemma are fulfilled for $\tau = \beta.\langle c.v\rangle$, $t^{pid} = \alpha|_{E^{pid}}$ (for $pid \neq pid_c$), $t^{pid_c} = \alpha^{pid_c}$, and $\lambda = \alpha|_V$.

Secondly, assume $pid_c \neq pid_v$. A trace $\alpha^{pid_c} \in E^{pid_c *}$ exists with $\beta|_{E^{pid_c}}.\langle c\rangle.\alpha^{pid_c} \in Tr^{pid_c}$, $\alpha^{pid_c}|_V = \alpha|_{V^{pid_c}}$, and $\alpha^{pid_c}|_{C^{pid_c}} = \langle\rangle$ hold because $BSD_{\mathcal{V}^{pid_c}}(Tr^{pid_c})$ (delete $\alpha|_{C^{pid_c}}$ in $(\beta.\alpha)|_{E^{pid_c}}$) and $BSI_{\mathcal{V}^{pid_c}}(Tr^{pid_c})$ (insert c after $\beta|_{E^{pid_c}}$). The assumptions of the Zipping Lemma are fulfilled for $\tau = \beta.\langle c.v\rangle$, $t^{pid} = \alpha|_{E^{pid}}$ for $pid \neq pid_c$, $t^{pid_c} = \alpha^{pid_c}$, and $\lambda = \alpha|_V$.

In both cases, $\alpha' \in E^*$ exists with $\beta.\langle c.v\rangle.\alpha' \in Tr$, $\alpha'|_V = \alpha|_V$, and $\alpha'|_C = \langle\rangle$. □

Proof. [of Lemma 12] Assume $\alpha, \beta \in E^*$, $c \in C^{pid} \cap \Upsilon^{pid}$, and $v \in V^{pid} \cap \nabla^{pid}$ with $\beta.\langle v\rangle.\alpha \in Tr^{pid}$ and $\alpha|_{C^{pid}} = \langle\rangle$. Hence, $c = trans(cid, val)$ where $dom_{ch}(cid) = high$, $receiver(cid) = pid$, and $sender(cid) \neq pid$.

We have $v = trans(cid', val')$ with $dom_{ch}(cid') = low$, $receiver(cid') = pid$, and $sender(cid') \neq pid$. $BSI_{\mathcal{V}^{pid}}(Tr^{pid})$ ensures that $\alpha' \in E^{pid*}$ exists with $\beta.\langle v.c\rangle.\alpha' \in Tr^{pid}$, $\alpha'|_{V^{pid}} = \alpha|_{V^{pid}}$, $\alpha'|_{C^{pid}} = \langle\rangle$. The event c has no preconditions and affects only $inbuf(cid)$. Since $cid' \neq cid$, $trans(cid', val')$ neither depends on $inbuf(cid)$ nor affects it. Hence, c and v can be exchanged in the trace, i.e., $\beta.\langle c.v\rangle.\alpha' \in Tr^{pid}$. □